# UNIVERSIDAD COMPLUTENSE DE MADRID

FACULTAD DE INFORMÁTICA

DEPARTAMENTO DE ARQUITECTURA DE COMPUTADORES Y AUTOMÁTICA



## TESIS DOCTORAL

**Aceleración de técnicas de ajuste de bloques mediante el procesador Nios II**

**(Nios II microprocessor-based acceleration of block-matching techniques)**

MEMORIA PARA OPTAR AL GRADO DE DOCTOR


PRESENTADA POR

**Diego González Rodríguez**

Director

Guillermo Botella Juan


**Madrid, 2014**

**UNIVERSIDAD COMPLUTENSE DE MADRID**

**FACULTAD DE INFORMÁTICA**

**Departamento de Arquitectura de**
**Computadores y Automática**


# TESIS DOCTORAL

# Aceleración de técnicas de ajuste de bloques mediante el procesador Nios II (Nios II microprocessor-based acceleration of block-matching techniques)

MEMORIA PARA OPTAR AL GRADO DE DOCTOR

PRESENTADA POR:

**Diego González Rodríguez**

Director:

**Guillermo Botella Juan**

**Madrid, Septiembre 2014**

# Aceleración de técnicas de ajuste de bloques mediante el procesador Nios II (Nios II microprocessor-based acceleration of block-matching techniques)


## TESIS DOCTORAL

*Memoria presentada para obtener el grado de*

*Doctor en Ingeniería Informática*

**Diego González Rodríguez**

*Dirigida por el profesor*

**Guillermo Botella Juan**

Departamento de Arquitectura de Computadores y Automática

Facultad de Informática

Universidad Complutense de Madrid

Madrid, Septiembre 2014

**Aceleración de técnicas de ajuste de bloques mediante el microprocesador Nios II**
**(Nios II microprocessor-based acceleration of block-matching techniques).**

*Memoria presentada por Diego González Rodríguez para optar al grado de Doctor por la Universidad Complutense de Madrid en el programa de doctorado en Ingeniería en Informática, realizada bajo la dirección de Guillermo Botella Juan.*

*Madrid, Septiembre de 2014*

*A mis padres y hermana*

*Y a Bea*

# Agradecimientos

Antes de nada, quisiera agradecer en primer lugar esta tesis al profesor Dr. D. Guillermo Botella Juan por el empuje incondicional que me ha proporcionado en cada momento de este trabajo. Quisiera agradecerle todo lo que ha hecho por mí, sin esperar recompensa alguna, la cantidad de horas que ha dedicado a esta tesis, toda la ayuda que me ha proporcionado, todos los momentos que ha sacrificado por mí, y sobre todo, el apoyo que me ha brindado tanto en el trabajo, como compañero y amigo.

Al Dr. Uwe Meyer-Baese de la Florida State University por su colaboración en este proyecto.

A mis padres por la educación que me han dado, el afán de superación que tienen, y el apoyo que me han dado en cada instante. Muchas gracias a los dos por llevarme hasta aquí, por haber hecho de mi lo que hoy he llegado a ser, y sobre todo por mostrarme siempre tanto cariño y hacer que os sienta siempre tan cerca.

A mi hermana, por los momentos que hemos pasado juntos y por su apoyo.

A Beatriz, por el apoyo que me ha dado durante todos estos años, la paciencia que ha tenido al no tener pareja durante muchos días, y todo el tiempo que he dejado de pasar con ella. Gracias también por aguantar mi mal genio durante la tesis.

A pipi, por todas las horas que estuviste a mi lado mientras hacíamos esta tesis, y la compañía que me brindaste durante tantos años.

A los FSN, por todas las risas y años que hemos compartido, por todos los buenos momentos que hemos pasado, y por los que nos quedan.

A mi familia, en especial a mi primo Jesús y a mis abuelos, a ti primo porque no puedas estar aquí, y a vosotros abuelos por qué aunque sea ley de vida se os echa de menos. Gracias sobre todo por los veranos que pasamos juntos.

A todas las personas que no he nombrado y que me han ayudado de una forma u otra.

Muchas gracias a todos.

# Abstract

This thesis deals with many techniques to accelerate motion compensation routines based on matching approaches. These routines comprise the most expensive and heavy part of many video coding standards, such as H.264, that have been evaluated, analyzed, and profiled. For achieving this, an exhaustive viability study of three very well-known matching-based motion compensation algorithms, widely used in multimedia coding, has been performed. These algorithms have been implemented in a low-cost FPGA using on the one hand three different architectural approaches, and on the other hand several algorithmic features.

The first technique deployed is based on Altera C2H compiler which reduces the peripheral hardware design complexity, enhancing the development of a SoC (System on a Chip). The second approach optimize the performance using an efficient combination of On-chip memory and SDRAM regarding the reset vector, exception vector, stack, heap, read/write data (.rwdata), read only data (.rodata), and program text (.text) in the design. The third approach addresses the optimization of the algorithms referred previously with the incorporation of custom instructions in the Nios II ISA. These custom instructions will be designed as combinational and multi-cycle design, and an efficient combination of both methods is then developed to build the final embedded system.

The results of these approaches suggest the accomplishment of a small sensor which processes 50×50 @ 180 frames per second, meaning a real-time motion compensation for the common video coding QCIF format at 19 frames per second. The present work thus, opens the door to motion coding for the low-cost soft-core microprocessors, particularly the RISC architecture of Nios II implemented entirely in the programmable logic and memory blocks of Altera FPGAs. This thesis gets together contributions to different research fields like Computer Vision, Multimedia Coding, Block-Matching Techniques, and FPGA based Embedded Systems.

# Resumen

Nuestra motivación en este trabajo es acelerar la ejecución de algoritmos de estimación de movimiento, ampliamente utilizados en estándares de codificación de vídeo como el H.264, usando dispositivos de muy bajo coste basados en microprocesadores empotrados (*soft-core*). Gracias a los avances logrados en este trabajo, los diferentes dispositivos de bajo coste pueden ver incrementadas sus funciones en lo que a codificación y gestión de video se refiere. Para ser capaces de acelerar los algoritmos elegidos dentro del campo de la estimación de movimiento, hemos usado tres estrategias diferentes combinando adicionalmente dos de ellas.

La primera, es la aceleración de las principales funciones del algoritmo a través del compilador Altera C2H, consiguiendo la generación de un módulo de hardware externo al microprocesador que trabaja con éste y que representa el funcionamiento de la parte elegida a acelerar del algoritmo, aliviando y reduciendo la carga de trabajo del microprocesador. La segunda estrategia, es la combinación de los dos tipos principales de memorias disponibles dentro de la FPGA, SDRAM y On-chip, en los diferentes módulos necesarios como la pila o el montículo entre otros, para la ejecución de los diferentes algoritmos. La tercera estrategia, que se combina con la segunda propuesta, se basa en la adición de una nueva instrucción para el repertorio de instrucciones del microprocesador. Esta nueva instrucción diseñada a medida, representa la parte del algoritmo donde hay una mayor pérdida del tiempo de ejecución. Dicha instrucción personalizada, se presenta como una instrucción monociclo en una primera versión y como una instrucción multiciclo en una versión posterior más avanzada.

Los resultados obtenidos como consecuencia de estas técnicas ponen de manifiesto la viabilidad de un sensor de bajo coste basado en el microprocesador Nios II que es capaz de procesar tiempo real para 50×50 @ 180 fotogramas por segundo, permitiendo compensación de movimiento para el formato multimedia QCIF a 19 fotogramas por segundo. En conclusión, este trabajo de investigación abre la puerta a la codificación de movimiento para microprocesadores Nios II con *soft-core y* coste reducido. Este trabajo presenta contribuciones a distintos campos de investigación como el de Visión por Computador, Codificación Multimedia, y Sistemas Empotrados basados en FPGA.

# Index

# Figures Index

# Tables Index

# Chapter I

## Introduction

This chapter is the introduction of the Ph.D. thesis. Its main objective is to motivate this work. Besides, it describes both computer and machine vision paradigms, as well as the FPGA architecture.

Moreover, the structure of this document and the publications derived from the Ph.D. thesis are shown.

## 1.1. Introduction.

In the modern era, the evolution of Mankind tries to endow machines, through the computerization process, of a higher similarity to humans so that machines can help in a better way and develop a larger amount of tasks. To achieve this goal, machines are being equipped with similar senses to humans, easing their interaction with the world around us.

One of the main senses which have been more advanced during the last years is the sense of vision [1 – 5], trying to gift machines the ability that humans have for reading the environment and extracting a big amount of information. This interest of the human race in producing machines with senses like sight is given by the easiness and the speed with which a machine can be gifted to analyze the environment information. In order to allow machines to manage the environment information, the first thing is to try to represent this environment, which is done thanks to cameras that obtain this information as images. An image is a visual representation which manifests the environment's visual appearance, an object or a group of objects which want to be portrayed in the image. Once solved this first drawback, we need to capture the environment information in a time interval larger than a moment. This issue has been solved thanks to video, which consist on a series of consecutives captures (images) with a small time interval between them, providing the environment information during a period of time.

Once video is provided (of images or frames) the next step towards our environment information analysis goes through codification, that is to say, the transformation of visual images in a set of analyzable data easily storable and therefore able to be communicated. Video codification [6 – 8] allows us to analyze easily this data set with machines, in which we have codified a video, either in the receiver system, which recorded that video, or in another system to which we would have to transmit that video, either in real time or in a time previous to its analysis.

One of the key points for video coding and therefore for its transmission too is the motion estimation, which is used to codify an image portion movement respect to another or to a group of these.

Insomuch as the need of transmitting larger amounts of video is bigger day by day, video coding standards increasingly try to erase the storage of redundant information between the different images that form the video, thanks to the motion estimation between these images, and thus transmit a larger amount of video with the least possible data size.

Due to the quick growth of the need of transmitting multimedia information around the world, motion estimation takes a key role in current technology, and this is why there are numerous video coding standards made up to this moment, and their evolution is continuous.

Inside motion estimation, there is a large number of algorithms commented in depth at Chapter 2, which have been developed to achieve the best estimation in the least possible time as they will be addressed in the next chapter. But inside this nanoseconds war, we find other key factors like the reliability of the estimation and therefore the algorithm result, which influences directly the codification process and therefore the video decoding.

This work is not focused in the possibility of creating a new motion estimation algorithm, but in the need of optimizing existing algorithms for their later integration inside real systems. So much so that which is presented in this work is not only an acceleration of motion estimation algorithms, but it is imprinted a guided structure to optimize and accelerate any algorithm without the need of working with motion estimation algorithms.

Regarding to motion estimation algorithms acceleration, there are numerous previous works as they are analyzed in the next chapter, so the main doubt is about which hardware has to be chosen for research. You can choose parallel processing, either by process parallelizing over a common processor, or process parallelizing to translate the acceleration to a GPU (Graphic Processing Unit) [9] device, as currently almost any hardware device with a CPU (Central Processing Unit) has available an adjacent GPU processor [10].

On the other hand, which is the one chosen in this work, we can speed up the algorithms in use through execution improvements, either combining the available

memory types in a hardware device, or through the addition of complementary hardware modules which avoid overloading the processor with every needed computation for the algorithms execution, or customizing the processor through new instructions added to the processor's instruction set which results in a lower number of clock cycles for the most common operations inside the execution of the accelerated algorithms.

Inside the available hardware devices to perform this "customization" of the algorithm, we have selected the FPGA (Field Programmable Gate Array)[1] due to the easiness of designing hardware circuits for it and its wide use in the research world, which makes simpler the development of this work and its integrations in real applications.

Once chosen the hardware device for the elaboration of this work, we have to face to another decision: this work's approach. This approach is divided into high performance systems, like for example supercomputers, and low cost devices in which energy consumption takes a vital role, as well as the number of used logic gates which impact directly both in the energy usage, and the needed hardware area for its placement.

This work takes the latter due to the growing need for a longer battery life of small mobile devices such as smartphones, tablets or simply video transmitter/receivers.

## 1.2. Motivation.

In the current world, performance in computation has become a very important asset. This time period which we are living in computerization history manifests directly in the execution time of any procedure, and so much so that movement estimation algorithms are suffering a continuous evolution towards faster executions through different types of accelerations.

Given the wide use of reconfigurable hardware systems like FPGA and the easiness of designing and reconfiguring hardware systems, we have chosen as the physical base of our work an Altera FPGA, which is one of the main brands used by researchers in the field of hardware systems design.

[1] FPGA definition: http://www.altera.com/products/fpga.html.

Our motivation for this work is accelerating and protecting the execution of motion estimation algorithms used in video coding standards, using very low cost devices based on embedded microprocessors (soft cores), so that thanks to the advances achieved in this work, the different low cost devices can see their performance increased when regarding to video codifying and managing.

For being able to accelerate the chosen algorithms within the available in the motion estimation algorithms world, we use three different strategies, and two of them mixed each other.

The first one is the acceleration of the main functions regarding the spent execution time through a so called Altera C2H (C to Hardware) Compiler[2], which operation is based in the creation of a hardware module outside the microprocessor but working together with this and reflecting the operation of the algorithmic part chosen, relieving the microprocessor from its execution and therefore reducing the microprocessor workload. The second strategy is the combination of the two main memory types available inside the FPGA, which are SDRAM (Synchronous Dynamic Random Access Memory) and On-chip, in the different needed modules for the execution of the algorithms, like the stack or the heap among others.

This second strategy will be combined with the third proposed strategy, which is based on the addition of a new instruction to the microprocessor's instruction set. This new custom instruction, designed for being executed using the least number of clock cycles, represents the algorithm part where there is a higher time leak. This custom instruction is presented like a monocycle instruction in a first version and like a multi-cycle instruction in a more advanced version of our custom instruction.

Furthermore, as accelerating paradigms, the evaluation of lexical obfuscation methods for common operations will be also presented, which is widely and intensively used in motion estimation for embedded system. In this part, we will evaluate Open source C, VHDL, and Verilog tools developed and tested for Altera, Xilinx tools and ARM devices, claiming that lexical obfuscation method is a viable IPP (Intellectual

[2] C2H: http://www.altera.com/literature/ug/ug_nios2_c2h_compiler.pdf.

Property Protection) method for multimedia coding and transmission protection, especially given the small penalty and the success of avoiding reverse engineering.

In addition to the time competition there is the size competition that increases the number of transistors by size unit allowing devices with better features to remain the same size or even a lower size than older devices. This second point of view, inside the competition between devices, has been the second boost in our motivation, doing this work has been focused on low cost systems, and therefore towards the lowest use as possible of the number of logic gates used for hardware systems designed to codify video.

On the other hand, increasing video resolution, which is every time higher, is translated directly in the number of needed bits to encode a video and therefore in the needed size for its codification. This resolution increase takes with it the process of a larger amount of data and therefore a higher execution time spent in the encoding process; so much so that the motion estimation acceleration appears again like an aid and a motivation to reduce the execution time spent in processing high resolution videos, avoiding the redundant data codification between the different images, and therefore its repercussion in the encoded video size.

## 1.3. Computer vision and machine vision.

Computer vision is the field of knowledge in which images are acquired, transformed into data, processed, analyzed, and understood in order to extract information from the environment. This image understanding is the extraction of information from the image data using a large amount of models based on geometry, mathematics or statistics. The input image data can be captured in many ways like video sequences, stereo video, or multidimensional video, and over this data, different theories and models are applied for the construction of a computer vision system.

Computer vision can be divided and classified into different categories. Depending on the granularity of the problem which is being treated, there are three classes in which computer vision can be classified [8]:

- Low level process: this level treats directly with image's pixels, extracting from the input images their properties, like color, texture or border. At this level the input images are analyzed and it is used a low level construction and codification of the images.

- Middle level process: this level groups the different objects achieved through the low level analysis in order to extract shapes and regions among other characteristics. Inside this level we can find segmentation and movement.

- High level process: at this level, specific objects and events are recognized. This level interprets the previous levels' data based in models and knowledge.

Now, focusing on the application of computer vision systems, they are used for industrial processes, navigation systems, modeling environments and objects, or as input for several devices including artificial intelligence devices. And focusing on its applications, computer vision is used for reconstruction, object recognition through several models, or motion.

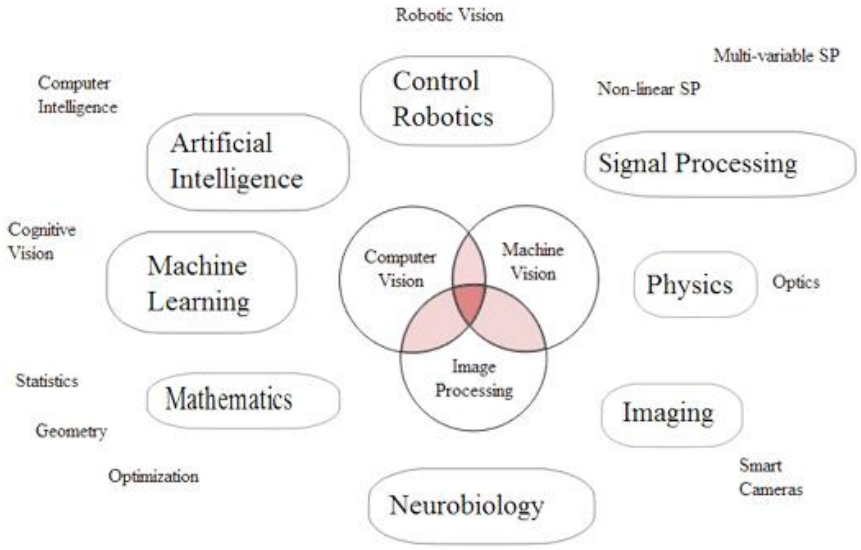


*Figure 1.1: Computer vision fields.*

Computer vision has a lot of related fields, in which it is used directly or indirectly, that have been presented in Figure 1.1 and described below:

- Artificial intelligence: it is the branch of computer science that studies and develops intelligent machines and software. Computer vision is used in an artificial

intelligence application as input, giving to the processing core the needed environment data [11].

- Machine learning: this is the branch of artificial intelligence which concerns the construction and study of systems which can learn from data. Again, computer vision is used for giving useful input [12].

- Mathematics: they are used at different levels for extracting information from the given images processed in the computer vision systems [13].

- Control robotics: computer vision is a very important element, controlling different processes like managing products in a production line [14].

- Signal processing: computer vision can be defined as a subfield of signal processing because signals given in the different images can be processed through computer vision [15].

- Physics: it is used in the capture process of a computer vision system, where there are sensors focused on detecting electromagnetic radiation. The application of Physics is key in the sensor design process [16].

- Imaging: this field is focused on the process of producing images. Also sometimes, this field analyzes images and processes them [17].

- Neurobiology: this field means the study of cells of the nervous system and the organization of these cells into functional circuits that process information and mediate behavior. The study of the biological vision system is very close to the computer vision [18 – 21].

As presented, computer vision is applied to a very large amount of fields. Think for example in a common application such as medical image processing, in which information is extracted from image data to make a diagnosis of the patient, and usually, these images come from microscopies, x-ray images, or ultrasonic images, and often are interpreted by humans after applying filtering or reducing the influence of noise. Another application area in computer vision is industry, where computer vision is used as an aid in the manufacturing process, making automatic the vast majority of several

processes. Between these processes we can find the quality control where, through computer vision, products are checked for defects automatically. A third application is in the military field where computer vision systems are mainly used in object detection and soldier recognition, including vehicles of every type.

The organization of a computer vision system is highly application dependent. Some systems are independent, solving by themselves the problem they are made to solve, but other computer vision systems are modules inside bigger systems, acting with specific function inside them.

The specific implementation of a computer vision system mainly depends on the specific application of the system, but although many functions are unique to the application, there are typical functions which are found in many computer vision systems. These main functions are described as follows:

- Image acquisition: a digital image is produced by one or several image sensors, which can be light-sensitive cameras or ultra-sonic cameras between others. Depending on the sensor type, the resulting image data is an ordinary 2D image, a 3D volume, or an image sequence. The pixels that compose the image have values that typically correspond to light intensity in one or several spectral bands (gray images or color images), but can also be related to various physical measures, such as depth for example.

- Pre-processing: before a computer vision method can be applied to image data in order to extract some specific information, it is usually necessary to process the data in order to ensure that it satisfies the method requirements, like for example, a noise reduction or a contrast enhancement.

- Feature extraction: image features are extracted from the image data using processes at different levels. These features are between others, lines, edges, corners, textures, or shapes.

- Detection/segmentation: this step is based in deciding which image points are significant, or/and which regions from the image are relevant for further processing with a specific set of interest points, or/and the segmentation of one or multiple image regions which contain a specific object of interest.

- High-level processing: at this step, in which we find image recognition, the input is usually an amount of data which should have implied a specific object.

- Decision making: in this step it is taken the final decision which the application is made for. For example, when working with pattern recognition applications, they should achieve a yes/no decision at this step.

While computer vision is used in many fields based on the automated analysis of images, machine vision goes beyond and combines automated image analysis with automated inspection and robot guidance. Machine vision is defined as the technology and methods used to provide imaging-based automatic inspection and analysis, for such it has a much extended scope, and inside this, we can find validation in quality processes, automatic control, or industrial robot leadership.

Machine vision can be started following a first step, which is the obtainment of an image that is usually achieved through lenses, i.e. cameras, like happened in computer vision. Although the vast majority of machine vision applications using like input two dimensional imaging, machine vision applications can use 3D imaging too. The imaging device, which obtains the images, can either be separate from the main image processing unit or combined with it, in this last case this combination is generally defined such a “smart camera” or “smart sensor”. On the other hand, when the image processing unit is separated from the processing core, the connection may be made through specialized intermediate hardware or a custom interface.

Following with the machine vision process, the second step would be the image processing. There is a plethora of methods for processing the obtained images, such as the following described below:

- Pixel counting: it counts the number of light or dark pixels.

- Thresholding: it results in a binary image taking as input its grayscale representation.

- Segmentation: to split an image into multiple image segments to analyze it easier.

- Pattern recognition: including template matching or automatic reading.

- Bar code reading.

- Edge detection: finding object edges.
- Filtering.

The third and last step in the machine vision process is the output of the whole process that usually is translated into final decisions. These decisions can be translated into exceptions which launch an alarm or notice. Moreover, this decision could be used for automatic systems as described for computer systems.

## 1.4. Consolidated accelerators: FPGA.

An FPGA (Field Programmable Gate Array) can be mainly described as a reprogrammable silicon chip which is made of prebuilt logic blocks and can be programmed thanks to routing resources. A FPGA can be configured for implementing custom hardware functionality without soldering or using a breadboard. A FPGA contains millions of connections and logic cells that can be configured to achieve a specific digital logic design. Instead of being restricted to any predetermined hardware function, a FPGA allows you to program product features and functions, adapt to new standards, and reconfigure hardware for specific applications, even after the product has been installed in the field, hence the name "field-programmable"[1].

FPGAs can be programmed in a large variety of low-level and high-level HDL (Hardware Description Languages) [22]. Due to the configurable capacity of the FPGA devices, a customized hardware can be designed to be included in any sensor. It is possible to design processor features, develop specialized hardware accelerators for intensive computation tasks, and create custom input/output ports to be connected with other physical part of the sensor. These systems, built together in the same FPGA, are nowadays known as SoPC (System on a Programmable Chip). FPGAs are configured developing digital computing tasks in software and compiling them down to a configuration file or bitstream that contains information on how the components should be wired together[3]. Thanks to these features, FPGAs achieve complete reconfiguration, and are able to fit a new design only with recompiling a different circuitry configuration.

[3] Introduction to FPGAs: http://www.ni.com/white-paper/6984/en/.

Due to its structure, an FPGA is very suitable to implement ASIC (Application Specific Integrated Circuit)[4] due to the easiness of reconfiguring its circuitry. This easiness presents many advantages for implementing applications. FPGA chip devices are well received in the different industries world around thanks to the combination between ASICs and processor-based systems which FPGAs offer. FPGAs provide both reliability and hardware-timed speed, but without needing very high volumes to warrant the high cost derived from custom ASIC design. Regarding processors, FPGAs are really parallel in nature, because of different executed operations do not have to race for the same hardware resources. Indeed, every independent execution task could work isolated into a dedicated section of the chip without influences from other logic blocks. As result, the performance achieved in one part of the application is not affected when adding more processing units.

FPGAs present some advantages over ASSPs (Application Specific Standard Product) or ASICs including rapid prototyping, shorter time to market, or the reprogramming feature between others.

Many years ago, FPGA technology could be only used by engineers with a deep knowledge of digital hardware design, but nowadays, thanks to the restated high-level hardware design tools, FPGAs are nearer to many people. Indeed, we could translate graphical block diagrams or even C code directly into digital hardware circuitry. Older generation FPGAs used I/Os (Input/Output) through programmable logic and interconnects, but new generation FPGAs are made with configurable embedded SRAM (Static Random Access Memory), high-speed transceivers and I/Os, logic blocks, and routing[1].

Summarizing the details, FPGAs are based on programmable logic components called LEs (Logic Element) which are the base units. These LEs are physically connected through reconfigurable connections which are the aim of the FPGAs. Thanks to the interconnections, when programming a FPGA we can achieve complex hardware designs, or simply logic gates like OR or AND. Moreover, the majority of FPGAs also contain memory elements in the logic blocks, either flip/flops or more complete blocks of memory. Because of the extended use of FPGAs, the devices have become more

[4] FPGA Vs ASIC: http://www.xilinx.com/publications/prod_mktg/easypath-7-fpga-asic-approach.pdf.

integrated. Indeed, due to Hard IP (Intellectual Property) blocks, which are built into the FPGA fabric, richer functions are provided. This is directly translated into lower power consumption. More evolved FPGA families, which include hard embedded processors, are very suitable to transform the devices into SoCs (System On a Chip) as we explain further in Chapter 4.

### *1.4.1. FPGA architecture overview.*

The most common FPGA architecture is, as described, made by LEs (Logic Element) that are the basic units, connected through interconnected reconfigurable channels, which usually have the same number of wires for this architecture. This wired connection will be performed according to a logic function specified by a user defined code. Because logic blocks are positioned as an array, multiple I/O pads may fit into the height of one row or the width of one column in this array, mapping the designed circuit into the suitable FPGA logic resources.

- ***LEs (Logic Element).***

Regarding to LEs, they are usually made up of two different components, flip-flops and LUTs (Look Up Table), connected by different ways depending on the FPGA family. When designing a hardware system to fit into a FPGA, the number of logic elements can be quickly decided, but the number of interconnections can oscillate, even among designs with the same amount of logic. We can see in Figure 1.2 a Cyclone II device LE, and a (Stratix II) device LE, containing each one four (eight) input LUT, one (two) register, and a carry in/out logic (two full adders) for use in arithmetic mode. LEs can be applied either for sequential and combinational functions. Each LE's programmable register provides several signals like data, clear, clock enable and clock input, being able to be configured for SR, JK, D, T modes [23]. Moreover, each LE can be used to implement combinational, sequential logic functions and arithmetic operations. Logic functions are implemented using a LUT instead of logic gates. Every LUT models any network of gates, defining a combinational logic function with many inputs (depending on the FPGA device) and one output. For combinational functions, the LUT output bypasses the register and drives directly to the LE outputs.

It is possible to connect more than one LE chained in the so-called LAB (Logic Array Block), through local interconnect channels, including each one 16 logic elements. When a logic function needs even more logic resources, the LABs can also be connected using the interconnection routing channels [24].

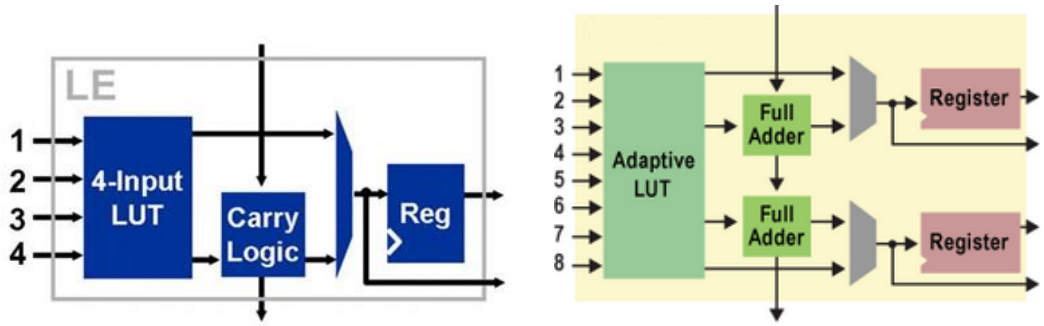


*Figure 1.2: Cyclone II logic element (left) [24]. Stratix II logic element (right)*[5].

Each LE can be configured in two modes: in the normal one (standard logic mode) or in the arithmetic mode [24]. The normal mode is used to implement combinational or sequential logic functions, nevertheless for efficient implementation of arithmetic functions is used the arithmetic mode, where the LE implements a 2 bit full adder and basic carry chain. This last mode is commonly used for implementing adders, counters, accumulators, and comparators.

- ***Embedded multipliers.***

Implementing a multiplication operation on an FPGA using standard logic blocks is not profitable, both in terms of resource usage and performance. On the other hand, multiplication operations are involved in most of the computer vision applications. Therefore, a way to implement high performance multiplication operations in some FPGA families, including Cyclone II, is to use hardware embedded multiplier blocks. These blocks can be used to multiply both signed and unsigned operands up to 250MHz performance. Each embedded multiplier can be used in one of two operational modes, which are single 18x18 bits mode, or two 9x9 bits mode, depending on the needed.

- ***Embedded memory blocks.***

Embedded memory blocks are internal RAMs (Random Access Memory) that provide a high throughput data access with low capacity data storage. The embedded

[5] Stratix II: http://www.altera.com/devices/fpga/performance/logic/per-logic-efficiency.html.

memory in Cyclone II devices consists of columns of M4K memory which can operate up to 250MHz. Each M4K block has a capacity to store 4608 bits including parity bits for error correction. They can be used to implement various types of memory including single-port RAM, ROM, FIFO (First In/First Out), simple and true dual-port with or without parity, and buffers. The word length can be adjusted between 1 and 36 bits. The memory content can be initialized by a user defined memory initialization file in programming state.

In a FPGA, each element is interconnected by a matrix of wires and programmable switches. By specifying the logic function for each logic element and selectively closing the switches, in the interconnection matrix will be implemented the desired design. The user defines the function usually written in HDL (Hardware Description Language) such as Verilog or VHDL. The logic function defined by the user is firstly converted to RTL (Register Transfer Level) format by a synthesizer tool. After that, it is generated by the fitter tool: a configurator for implementing this RTL structure by FPGA LEs, specifically for each device used.

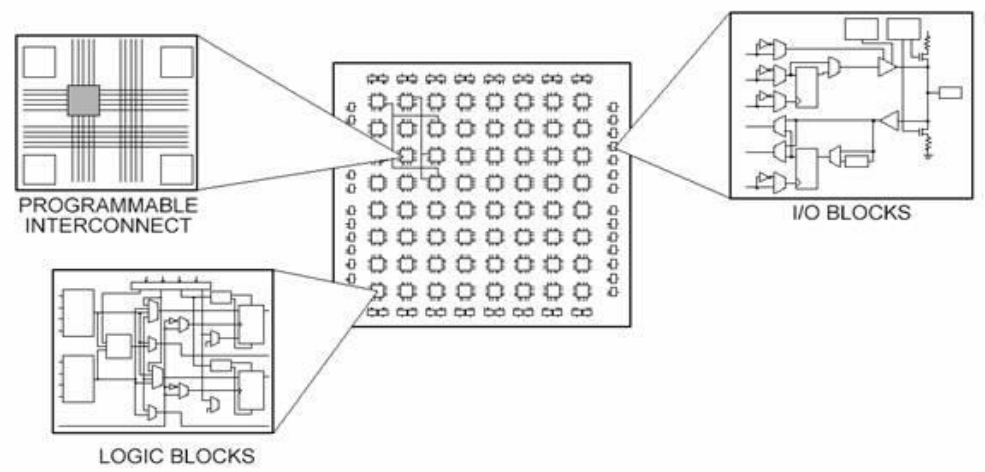


*Figure 1.3: The different parts of an FPGA[6].*

For example, a systolic array and a crossbar switch take the same amount of logic elements, but both do not require the same routing interconnections. Because unused interconnections do not provide any benefit and degrade the performance, FPGA manufacturers attempt to provide just enough routing interconnections so that the majority of designs will fit in terms of LUTs and I/Os that can be routed.

[6] FPGA fundamentals: http://www.ni.com/white-paper/6983/en/.

FPGA logic blocks usually contain several logic elements as presented before. Like shown in Figure 1.2, FPGA manufacturers started to use 8-input LUTs in their high performance parts to increase the FPGAs performance. Focusing on interconnections, specific signals like clock events among others are often connected through dedicated routing tracks.

Moreover, many FPGAs do not have its routing segmented. In FPGA architectures based on switch boxes this means that every wiring segment reach a unique logic block previously passing through a switch box. We can achieve longer tracks programming some switches from a switch box. Indeed, when it is needed a higher speed interconnection, many FPGAs use routing tracks which perform multiple logic blocks.

These switch boxes are presented wherever both vertical and horizontal channels intersect. Depending on the FPGA architecture, there are several ways of programming switches to provide interconnections with other wires from contiguous channel segments. In Figure 1.4 (left), we present an example of FPGA architecture using switch boxes for interconnections.

Moreover, it is shown in Figure 1.4 (right) the FPGA programming plane, which gathers the LUT values in SRAMs for configuration of functional plane at start up, and the functional plane, which implements logic functions defined by the user through routing channels and logic cells.

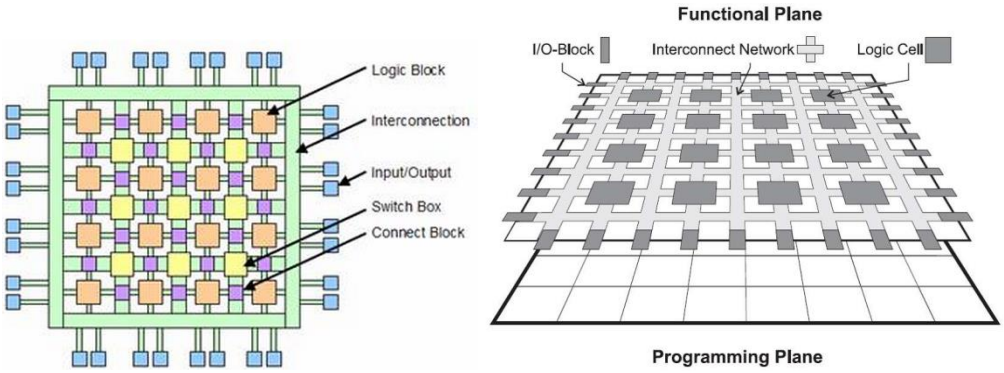


*Figure 1.4: Example of FPGA architecture using switch boxes for interconnections (left)[7]. FPGA programming and functional planes (right).*

One accepted classification of FPGA architectures considers both functional and programming planes. Programming plane stores values in SRAM (Static Random

[7] FPGA architecture: http://ca.olin.edu/2005/fpga_dsp/fpga.html.

Access Memory) structures which define the LUTs values, connect switch states, and configure I/O blocks for pin direction selector. Nevertheless, LEs and interconnect routing channels are located in the functional plane. Regarding the configuration of the device used in this work (Cyclone II), it is configured by loading the configuration program to internal SRAM. Because of the usage of SRAM, the configuration will be lost whenever power is removed. In actual systems, a small external low-cost serial flash memory or PROM (Programmable Read Only Memory) is usually managed to automatically load the FPGA's programming information when the device powers up.

### *1.4.2. Device selection classification.*

Considering the criteria to choose the device for computer vision applications, it is crucial to define strictly the application area. For example, a transceiver design for high speed communication applications needs different resources than an image processing hardware design. Another aspect is the amount of needed resources, which varies depending on the complexity of the design. FPGA development boards are available in a huge range of sizes with different feature sets. Diverse boards contain many set of I/O interfaces, logic, memory, and other hardware. Provided an FPGA board which has enough logic resources and the needed peripherals, a project can be implemented there. As a general rule, FPGAs with more I/O interfaces, more memory, higher speed, and more peripherals, are more expensive. Due to this, choosing the right board when making the selection, and having the FPGA chip the proper feature set at lower cost, is a key consideration.

That board must have low power consumption and moderate performance since it will be used in the scope of autonomous and robotic platforms. Additionally is the capability of the board to add future modifications of the design improvements. As remarked previously, each board has a different number of LEs used for implementing logic. They also contain many amounts of external memory devices such as RAM and ROM (Read Only Memory) memory chips, with larger capacities than the internal memory but having a lower bandwidth and slower access times.

As usual in the motion estimation field, the algorithm implementations process a large amount of data. An algorithm that processes a single frame at each time step can be implemented easily in a hardware that processes input data sequences without storing

the whole data in memory. However, a motion estimation algorithm needs two or more frames from consecutive time steps. This requires the whole images to be gathered in memory before processing. The massive amount of data prevents it to be stored in internal memory blocks that are deemed insufficient. Therefore, external memories such as SDRAM (Synchronous Dynamic Random Access Memory) or SSRAM (Synchronous Static Random Access Memory) must be used, although they have lower bandwidth. In terms of throughput, SSRAMs are faster but have lower capacity compared to SDRAMs. However, to prevent memory bottleneck, SSRAMs should be adopted for frame buffering.

Motion estimation computation includes intensive multiplication operations, and the implementation of multiplication by programmable logic is not preferable both in terms of performance and its high usage of resources. To solve this drawback, there are some FPGAs available which support DSP (Digital Signal Processing) applications, containing thus, higher performance based hardware integer multipliers at no expense of logic resources.

The final design throughput is proportional to the clock frequency operation, and the limiting factor of the clock rate is so-called critical path, which is the highest delay between the registers. In order to decrease the limiting effect of that handicap, the higher delayed paths are clocked at lower rates and the parts of the design with lower delays are clocked at higher rates. This technique is called the multiple clock domain design. Therefore, the design works and consequently needs multiple clock frequencies. Each board contains a crystal controlled clock circuit that is usually used as the master clock for the user's digital logic circuit. Additionally, PLL (Phase Locked Loop) circuits can be used to scale the crystal controlled clock to provide other clock frequencies. There are multiple PLLs utilized to multiply or divide the clock frequencies and shift the phase of clock signals to generate different clock signals.

To send and receive data between the FPGA board and the PC (Personal Computer), there should be many communication interfaces ready to use on board. This communication is usually established through either USB (Universal Serial Bus), Ethernet or RS232 Protocol.

Regarding pins, FPGAs provide a wide variety of I/O features in terms of I/O voltage, pin count, and pin drive strength. The number of pins must be enough to drive the external devices, and it is generally helpful to have general purpose I/O pins for user access. Many of the peripherals need controllers for interfacing with the FPGA logic and it is desirable to have hardware controllers provided on board. Additionally, logic that provides a device interface circuit or controller will need to be constructed by the user, using the FPGA's internal logic resources. Switches, buttons, seven segment displays, LEDs, and LCD displays, although not obligatory, become really helpful both for debugging the hardware design process, and for the FPGA user interface.

### *1.4.3. FPGA Development platform specifications.*

The choice of the FPGA hardware development board is achieved according to the requirements explained before.

There are two main vendors, Altera and Xilinx, which provide many suitable boards. Moreover, there are two categories of FPGAs from each vendor, called the "low cost" and "high end" devices. We have considered as suitable boards those which include FPGA chips classified as low cost devices (moderate performance and low power). The FPGA product portfolio of Altera is shown in Figure 1.5 including a comparison of the available performances, resources, costs, and power consumptions. Although they promise a high performance operation, high end FPGA devices such as Stratix V and bigger, are not feasible for our needs because of the high power consumptions and difficult portability to autonomous mobile robotic platforms. Therefore, it has been chosen a device that is in the low power devices category.

Altera names its low power devices as "Cyclone family". So, we choose a Cyclone II device due to its low power consumption, enough number of resources, and low cost.

| Device | Process | LE | Memory | DSP Blocks | DSP (MHz) |
|---|---|---|---|---|---|
| Stratix V | 28 nm | 1087 K | 50 Mb | 3510 | 550 |
| Stratix IV | 40 nm | 820 K | 23.1 Mb | 1288 | 550 |
| Stratix III | 65 nm | 338 K | 16.2 Mb | 768 | 550 |
| Stratix II | 90 nm | 180 K | 9.0 Mb | 384 | 450 |
| Stratix | 130 nm | 80 K | 7.3 Mb | 56 | 350 |
| Arria II | 40 nm | 350 K | 16.4 Mb | 1040 | 350 |
| Arria | 90 nm | 90 K | 8.5 Mb | 736 | 350 |
| Cyclone IV | 60 nm | 150 K | 6.3 Mb | 360 | 290 |
| Cyclone III | 65 nm | 120 K | 3.8 Mb | 288 | 290 |
| Cyclone II | 90 nm | 68 K | 1.1 Mb | 150 | 250 |
| Cyclone | 130 nm | 20 K | 0.2 Mb | - | - |

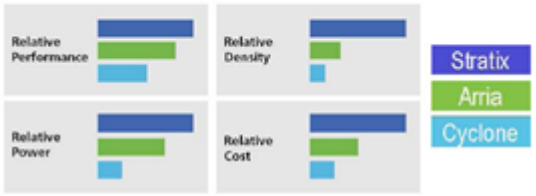


*Figure 1.5: Altera FPGAs product portfolio (up). Altera available FPGA families comparison (down).*

The peripheral devices such as memory are also crucial when choosing the platform, so as we will see in Chapter 6, SDRAM will be adopted because of its large capacity and low cost per MByte. For PC communication, we preferred USB due to its easy interface. However, for future improvements, a faster communication protocol such as Ethernet is also beneficial. The FPGA development board selected is DE2-C35, which is also widely used in many universities around the world for education and research purposes. The top view of the DE2-C35 board is given in Figure 4.11.

The FPGA chip included in the DE2-C35 board has average resources among the available devices in the Cyclone II family, named as EP2C35. Even so, it provides enough memory blocks, embedded multipliers, high logic elements, I/O pins, and PLLs, required for our current design needs at this thesis. The Cyclone II family features are given in Table 1.1.

| Feature | EP2C5 | EP2C8 | EP2C15 | EP2C20 | EP2C35 | EP2C50 | EP2C70 |
|---|---|---|---|---|---|---|---|
| LEs | 4,608 | 8,256 | 14,448 | 18,752 | 33,216 | 50,528 | 68,416 |
| M4K RAM blocks (4 Kbits plus 512 parity bits | 26 | 36 | 52 | 52 | 105 | 129 | 250 |
| Total RAM bits | 119,808 | 165,888 | 239,616 | 239,616 | 483,840 | 594,432 | 1,152,000 |
| Embedded multipliers | 13 | 18 | 26 | 26 | 35 | 86 | 150 |
| PLLs | 2 | 2 | 4 | 4 | 4 | 4 | 4 |
| Maximum user I/O pins | 158 | 182 | 315 | 315 | 475 | 450 | 622 |

*Table 1.1: Cyclone II FPGAs family features[8].*

### *1.4.4. FPGA advantages to implement machine vision systems.*

Once we have presented the bases of the FPGA architecture, we can detail and show the different advantages achieved when designing our motion estimation circuitry on FPGA architecture. These advantages comprise among others reliability and flexibility without setting aside the execution speed. Described below are the main benefits of FPGA technology:

- Performance: using their main feature, which is hardware parallelism; FPGAs present better computing power than DSPs (Digital Signal Processor), leaving behind the sequential execution and achieving better results per clock cycle. Moreover, through the control of I/Os at hardware level, it is achieved a faster response and a more customizable functionality which make them nearer to desired systems requirements.

- Time to market: due to the customizable FPGA architecture, these devices present a higher flexibility and faster prototyping capabilities regarding time-to-market concerns. Thanks to it, it is not needed to wait for the long fabrication process of a custom ASIC. Instead, using FPGA architecture, designers can check, verify, test and run their ideas in a short time. Moreover, designers are able to implement changes and design new functionalities on a FPGA design within hours instead of weeks. The increasing growth of high level software tools for designing under FPGAs is making the learning curve for these architectures decrease with the help of abstraction layers and prebuilt functions through IP (Intellectual Property) cores.

[8] Altera Cyclone II: http://www.altera.com/devices/fpga/cyclone2/cy2-index.jsp.

- Cost: the recursive manufacturing process when designing a custom ASIC is much more expensive than the recursive hardware design process under FPGA-based hardware solutions. Moreover, to get the return of the high cost in the ASIC manufacturing process, it is needed to sell thousands of chips per year, while in the opposite side, many end users need customize hardware designs in the development of systems. This means that FPGA architecture do not have extended cost during assembling process thanks to the nature of being programmable. Due to the fact that system requirements are continuously changing, the cost of taking these changes in FPGA architectures is lower by far than the high cost of implementing these changes under ASIC architectures.

- Reliability: FPGA architectures are an image of program execution through its hardware implementation, while in the opposite side software tools provide the programming environment for a fixed architecture. Indeed, systems based on processor usually carry several abstraction layers to schedule the different tasks and be able to share the hardware resources among multiple processes. These control duties are managed by the OS (Operating System) while driver layers provide hardware resources control. Single processor based systems only can execute one instruction at a time, and also have the risk of time critical tasks between the different execution threads. In the FPGA architecture, it is avoided the use of an OS. Moreover, they use true parallel execution, minimizing reliability concerns and providing dedicated hardware to every task.

- Long term maintenance: hardware systems based on FPGA architecture are easily upgradable and do not require so much time and so much cost derived from the changes as in a ASIC upgrading process. Focusing on continuous changing requirements, digital communication protocols for example have changing specifications, and ASIC-based interfaces would have several challenges derived from the upgrading process. FPGAs with its reconfigurable architecture can maintain future changes and upgrades as necessary. Indeed, future added functionalities can be implemented without altering the board layout.

Now, we know about how FPGA architecture logic blocks are configured with millions of components to achieve the hardware design which is needed to be executed.

This process consist in using development tools and a compiling process to get a configuration file or bitstream which holds the information on how logic elements should be configured and linked to achieve the desired hardware design. The process of fitting the configuration file into the current FPGA architecture is called "place and route". Before fitting the design, the end user can check the netlist and the "place and route" results through timing analysis or simulation between other methodologies. Once these previous steps are completed, it is generated a binary file to reconfigure the FPGA. This file is transferred to the FPGA through many ways such as using a serial interface JTAG (Joint Test Action Group).

FPGA programming is changing in the recent years thanks to the growth of HLS (High Level Synthesis) design tools, which make it easier and faster thanks to graphic block diagrams between other improvements. Older FPGA architectures were only programmed by designers with very good skills of hardware design.

During the first decades in FPGA development, HDLs (Hardware Description Language) based on dataflow models such as VHDL and Verilog look up to main software languages to design under FPGA architectures. These low level languages carry some advantages inherited by other textual languages, but with the difference that they are designing a circuit. This derives into a syntax which mixes low level and high level programming, and requires mapping and connection of signals from I/O ports to internal signals wired and used inside algorithm functions. These functions are sequentially executed and also can reference other functions inside the FPGA. However, the truly parallel execution in FPGA architecture is not easy to see within its sequentially executed functions.

Previously mentioned IP cores are defined like libraries which contain complex functions and have been previously tested and optimized to use as independent modules. Moreover, IP cores can be provided by FPGA vendors, third party suppliers, or developer communities which sometimes are open source.

### *1.4.5. FPGA development environment and design flow.*

As far as the higher gate densities and design complexity are growing up the designing methods are changing. Rapid prototyping using HDLs, IP cores, and logic

synthesis tools, have substituted the traditional gate level design using a schematic editor.

These new HDL-based logic synthesis tools can be used for both FPGA-based and ASIC designs, while the two most widely HDLs used are Verilog and VHDL. The hardware design process is a complex task which uses computational resources. However, in order to simplify the design process, FPGA vendors provide useful design tools which are integrated in a user friendly FPGA front-end supporting behavioral description based design using an HDL.

- ***Quartus II.***

The computer aided design software provided by Altera for FPGA designs is called Quartus II software [25]. This software provides the designer a number of tools which are used in the FPGA design process. All these tools are included in an integrated design environment with easy to use interfaces. Because of it, we use this package in the hardware design of this thesis, including tools like:

- HDL editor, Analysis & Synthesis, and Fitter tools.
- Classic & TimeQuest Timing Analyzer.
- Signaltap II Embedded Logic Analyzer.
- Powerplay II Power Analyzer.

- ***FPGA design flow.***

Actually, the FPGA design flow is based on design a logic circuitry that performs the desired behavior, being possible to define this flow differently. The simplest method is drawing the schematic diagram of the circuitry. However, for larger designs, the schematic diagram of the circuitry becomes very complicated, resulting this method not functional for large designs. Another generally preferred method is using a HDL to define the logic circuitry, existing two most popular HDL paradigms as commented previously.

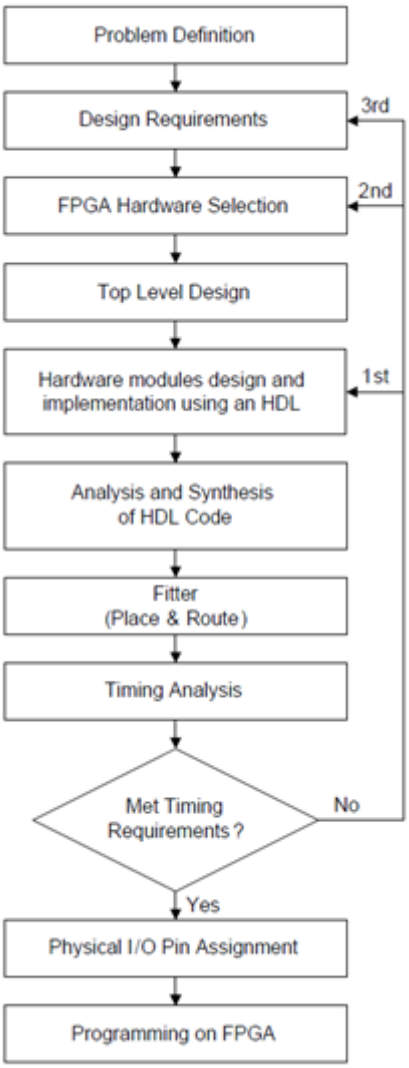


*Figure 1.6: FPGA design flow block diagram.*

Taking into account the hardware design, it has been adopted the top-down methodology shown in Figure 1.6, where high level functions are defined first, and the lower level implementation details are filled in later. This methodology simplifies the design task and allows the partitioning of the whole design into manageable subparts. The top level block represents the entire hardware structure and each lower level block represents major functions.

The first step is to make the problem definition properly and the main function of the design, where inputs and outputs of the system must be determined. In the second step, the designer should state the design requirements clearly, including constraints on the system throughput, power consumption, memory size, clock rate, weight, and hardware volume. So, the designer has to determine a suitable hardware according to the

requirements. Having defined the design requirements, the design is hierarchized into functional blocks, being those blocks designed, implemented using a HDL language, and tested individually.

The written HDL code is analyzed and converted into low level representation, which is called RTL (Register Transfer Level) description, by synthesis tools. After that, the generated netlist should be reformed to be placed into the specified hardware. This process is carried out by the specific fitter tools which optimize the logic according to the hardware resources available.

Later, the design is placed and routed for the device, and the timing requirements such as clock delay, setup and hold times, etc. must be checked. This process is so-called timing analysis. If the timing constraints were not met, the first action should be to revise again the hardware design. If alternative design approaches do not solve the problem, then the second response will be moving to another device with higher speed grade. If there were no device capable of implementing the design, this would mean the design is not feasible with the current specifications.

Therefore, the design and timing constraints have to be checked whether they can be satisfied. When timing requirements are met, the next step is to assign the physical device pins to the design inputs and outputs. The last step is the device programming.

Most of the current FPGA architectures do not include any on-chip program memory. So, the configuration is generally stored in an external EEPROM (Electrically Erasable Programmable Read Only Memory) memory and loaded into the FPGA SRAM on every power up.

## 1.5. Thesis organization.

The presented work is organized in seven chapters which are following described:

- Chapter 1: introduction to this work and motivation for this thesis. Also, we have defined in this chapter the basis of this work describing on the one hand computer and machine vision, and on the other hand FPGA architecture.

- Chapter 2: it describes the basis of the motion estimation, including the different algorithm families and their implementation in the real world.

- Chapter 3: it talks about block-matching, video standards, and the different algorithms inside the motion estimation family. This chapter is completed with other algorithms and, used error metrics, and image inputs.

- Chapter 4: this chapter takes a look into the embedded systems and their evolution through history. Also, soft and hard processors are shown centering on the Nios II Altera embedded processor. Some of the SoPC systems peripherals are described, and also their relation with the processor.

- Chapter 5: it is centered in the acceleration of block-matching algorithms using the Altera Nios II C2H Compiler, moving specific functions which are critical for performance, from running on the FPGA soft-core processor (Cyclone II EP2C35F672C6N) to optimized and pipelined hardware accelerators. These accelerators have direct access to the processor's memory, improving largely the parallel transactions to the needed number of buffers.

- Chapter 6: it describes the hardware acceleration using the combination of the different memory types in our FPGA hardware device and developing a custom instruction, which has been designed in two different ways, monocycle and multi-cycle.

- Chapter 7: it shown work conclusions and future work.

- Appendix 1: Additionally, in the framework of this research work it is presented at this appendix the evaluation of lexical obfuscation methods for common operations widely and intensively used in motion estimation for embedded systems. Open source C, VHDL, and Verilog tools, have been developed and tested for Altera, Xilinx tools, and ARM devices. Therefore, obfuscation is proved as a viable IPP method for multimedia coding and transmission protection, especially given the small penalty and the success of avoiding reverse engineering.

- Appendix 2: As summary, complete resume of this thesis in Spanish language is presented here.

## 1.6. Ph.D. Thesis publications.

This investigation has led into different publications such as 3 journal papers indexed at ISI (Institute of Scientific Information) JCR (Journal Citations Report) as Q2, Q3, and Q1, 1 international recognized conference which holds a low acceptance rate (26%), 1 book chapter, 2 international conferences, and 1 national conference. They are presented chronologically.

***J1*** *"Code obfuscation using very long identifiers for FFT motion estimation models in embedded processors"*, Uwe Meyer-Baese, Anke Meyer-Baese, **Diego González**, Guillermo Botella, Carlos García, Manuel Prieto, Journal of Real Time Image Processing (JRTIP) 2014, (In press). **(Impact Factor: 1.156. Position : 117/243, Q2)**

***J2*** *"Acceleration of block-matching algorithms using a custom instruction-based paradigm on a Nios II microprocessor"*, **Diego González**, Guillermo Botella, Carlos García, Manuel Prieto and Francisco Tirado, EURASIP Journal on Advances in Signal Processing 2013, 2013:118 doi:10.1186/1687-6180-2013-118. **(Impact Factor: 0.807. Position: 155/243, Q3)**

***J3*** *"A Low Cost Matching Motion Estimation Sensor Based on the NIOS II Microprocessor"*. **González, D**.; Botella, G.; Meyer-Baese, U.; García, C.; Sanz, C.; Prieto-Matías, M.; Tirado, F. Sensors Basel 2012, 12, 13126-13149. **(Impact Factor: 1.953. Position: 8/57, Q1)**

***C1*** *"Optimization of block-matching algorithms using custom instruction-based paradigm on NIOS II microprocessors"* **Diego González**, *Guillermo Botella, Anke Meyer-Baese, Uwe Meyer-Baese, , in Independent Component Analyses, Compressive Sampling, Wavelets, Neural Net, Biosystems, and Nanoengineering XI, Harold H. Szu, Editors, Proceedings of SPIE Vol. 8750, 2013 (SPIE, Bellingham, WA 2013), 87500Q.*

***C2*** *"NIOS II processor-based acceleration of motion compensation techniques".* ***Diego González****, Guillermo Botella, Soumak Mookherjee, Uwe Meyer-Bäse, Anke Meyer-Bäse, , in Independent Component Analyses, Wavelets, Neural Networks, Biosystems, and Nanoengineering IX, Harold Szu, Editors, Proceedings of SPIE Vol. 8058 2012(SPIE, Bellingham, WA 2011), 80581C.*

***C3*** *"FPGA-Based Acceleration of Block Matching Motion Estimation Techniques,"* **Gonzalez, D.**; Botella, G.; Mokheerje, S.; Meyer-Base, U., *Field Programmable Logic and Applications (FPL), 2011 International Conference on* , vol., no., pp. 389-392, 5-7 Sept. 2011doi: 10.1109/FPL.2011.76. Indexed Conference at www.cs-conference-ranking.org **0,82 (70/421 total conferences**

**and 24/57 top conferences):** Acceptance 2011 rate of FPL is 26%.

**BCh1** *"Real-Time Motion Processing Estimation Methods in Embedded Systems"*, Real-Time Systems, Architecture, Scheduling, and Application[9], Dr. Seyed Morteza Babamir (Ed.), ISBN: 978-953-51-0510-7, InTech, DOI: 10.5772/37789. Authors: Guillermo Botella and **Diego González** (2012). Chapter 13, pp 265-292.Capítulo de libro.

***NC1*** *"Diseño rápido y modular de microprocesadores para aplicaciones específicas (ASIP)*", **Diego González**, Guillermo Botella, Luis Parrilla. XI Jornadas de Computación Reconfigurable y Aplicaciones (pág. 149-154), Universidad de la Laguna. Septiembre 2011.

[9] Available from: http://www.intechopen.com/books/real-time-systems-architecture-scheduling-and-application/real-time-motion-processing-estimation-methods-in-embedded-systems-

# Chapter II

## Motion Estimation

This chapter serves as an introduction to the motion estimation paradigm, addressing several methods and techniques used for estimating motion in different contexts. Furthermore, it describes in detail the state of the art by analyzing a wide variety of implementations deployed in different platforms, such as: FPGAs, GPUs, and ASIC devices.

## 2.1. Introduction.

Motion estimation is a low level vision task, which nowadays controls a high number of applications as sport tracking, surveillance, security, industrial inspection, robotics, navigation, optics, medicine and so on. Unfortunately, most times it is unaffordable to implement a fully functional embedded system for real time operation working with enough accuracy due to the nature and complexity of the signal processing operations involved.

In the second chapter of this thesis we will introduce different motion estimation methods and their implementation when real time is required. The present chapter is organized as follows: after the introductory paragraph, the second section explains the Motion Estimation and Optical Flow paradigms, and their similarities and differences to understand the real time methods and algorithms. After that, a classification of motion estimation methods according to different families is presented, and enhancements for further approaches are performed. This section is finished with a plethora of real time implementations of the methods based on FPGA, GPU, and ASIC motion estimation implementations platforms, analyzing briefly the main pros and cons of these approaches, and their resource consumption.

## 2.2. Motion Estimation.

The term “optic flow” was coined by James Gibson after the Second World War, when he was working on the development of tests for pilots in the U.S. Air Force [26]. His basic theory provided the basis for much of the computer vision works 30 years later [27]. In later publications, Gibson defines the gradient of deformation of the image along the motion of the observer, a concept that was dubbed “optical flow” or simply “flow”.

When an object moves, its two dimensional projection image moves with the object as shown in the Figure 2.1. The projection of three dimensional motion vectors on the two dimensional detector is called the apparent field of flow velocities, or image. The observer does not have access to the velocity field directly, since the optical sensors provide luminance distributions, not speed, and motion vectors may be very different from this luminance distribution.

The term "apparent" refers to one of the biggest problems in computer vision, the absence of real speed, but a two dimensional field called motion field. However, one can calculate the movement of local regions of the luminance distribution, being known as the field of optical flow motion, providing only an approximation to the actual field of speeds [28].

As an example, it is easy to see that the optical flow is different from the velocity field, see Figure 2.2, where there is a sphere rotating with uniform brightness, which produces no change in the luminance of the image. The optical flow is zero everywhere in contradiction with the velocity field. The reverse situation yields with a static sphere with a moving light in which the velocity field is zero everywhere, even though the luminance contrast induces non-zero optical flow. Another example is the rotational poles announcing the barbershops of yesteryears, with the velocity field perpendicular to the flow.

Among these atypical situations, under certain limits it is possible to recover motion estimation. Motion algorithms described in this chapter seek to recover the flow as an approximation of the velocity field projected as flow information, being a two dimensional array of vectors that may be subject to several high level performances. In the current literature there are several applications in this regard [29 – 31].

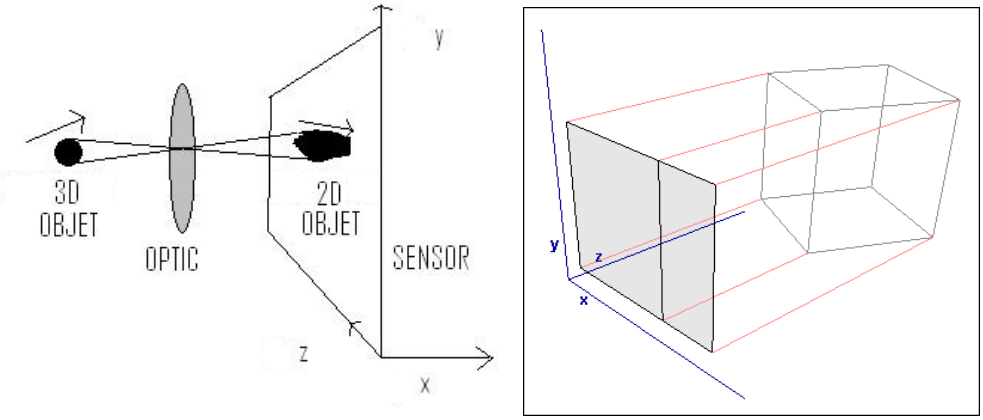


*Figure 2.1: Projection from 3D world to 2D detector surface [32].*

The interpretation and use of optical flow is an ill-conditioned problem since, as we have remarked, it is based on imaging in three dimensions on a two dimensional detector. This process removes information, and its recovery is not trivial. Therefore,

the flow is a measure of the ill-posed problem in itself, since there are infinite velocity fields that may cause the observed changes in the luminance distribution. In addition, there are infinite three dimensional movements which can generate a given field of velocity.

It is therefore necessary to take into account a number of considerations to restore the flow. The so called problem of aperture [33 – 35] appears when measuring the two dimensional velocity components using only local measurements. It is possible to recover the component of velocity gradient perpendicular to the edge, forcing adding external conditions that usually require obtaining information from a finite neighbourhood space, see Figure 2.3 (left). To resolve this problem, the region must be large enough to get a solution, as the search for a corner, for example. However, the collection of information across a region increases the probability of also taking on different motion contours and consequently to truncate the results, needing a trade-off solution. This last problem has been named as the aperture general problem [36].

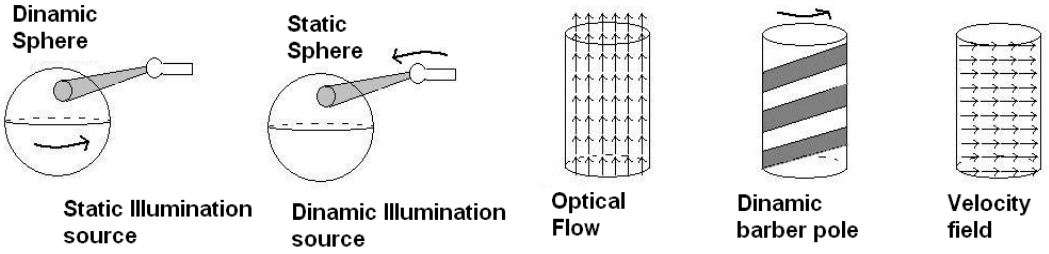


*Figure 2.2: Difference between velocity field and optical flow [37].*

### 2.2.1. State of the art estimating real time motion.

There are many algorithms and architectures for real time optical flow estimation which are frequently used, emanating from artificial intelligence, signal theory, robotics, psychology, and biology. Indeed, there is a vast literature about them, but it is not the purpose of this section to explain all the algorithms.

It will be reviewed the state of the art as descriptive as possible for the sake of clarity, in order to justify specifically the real time implementations presented at the end of this chapter.

We can classify motion estimation models into three different categories:

- Correlation based methods: they work comparing positions from image structure between adjacent frames and inferring the speed of the change in each location. Also, they are probably the most intuitive methods [38].

- Differential or gradient methods: these are derived from work using the image intensity in space and time. The speed is obtained as a ratio from the above measures [39 – 40].

- Energy methods: they are represented by filters constructed with a respond oriented in space time to work at certain speeds. The structures used for this type of processing are parallel filter banks, which are activated for a particular range of values [41].

The different approaches to motion estimation are appropriate under each application. According to the sampling theorem [42], a signal must be sampled at a sampling rate, which is at least twice the maximum frequency that has such signal. Therefore, it ensures us that in the field of moving image processing, movement between two frames is small compared to the scale of the input pattern.

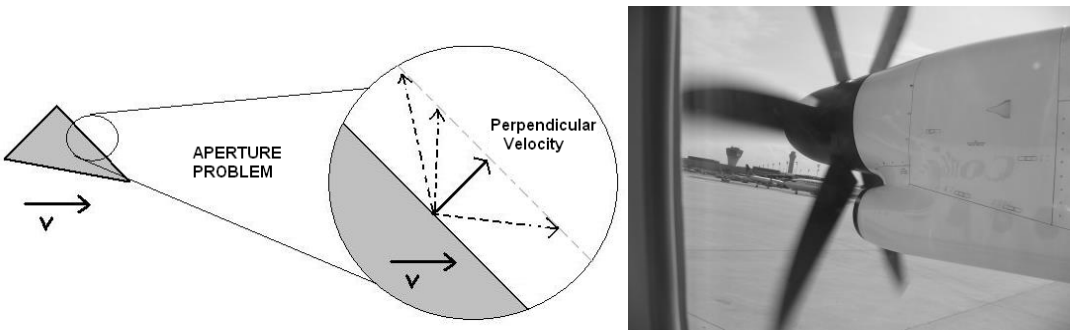


*Figure 2.3: Difference between velocity field and optical flow [37].*

When this theorem is no longer fulfilled, the phenomenon of sub sampling or aliasing appears. In space time images, this phenomenon produces false inclinations or structures unrelated to each other. As an example of temporal aliasing; we can observe a rotation of the propeller of the planes in the opposite direction to true as shown in Figure 2.3 (right). In short, no long trips can be estimated from input patterns with small scales. In addition to this problem, we have the problem of aperture, discussed

previously. These two problems, aliasing and aperture, form up the global problem of correspondence, as shown in Figure 2.3.

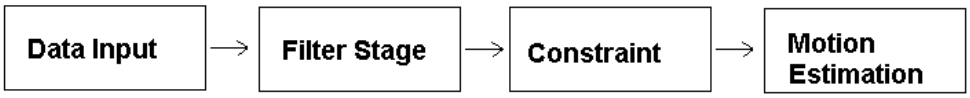


*Figure 2.4: Functional data flow commonly used for dealing with motion estimation.*

Therefore, the movement of the input patterns do not always corresponds to features of consecutive frames in an unambiguous manner. The physical correspondence may be undetectable due to the problem of aperture, lack of texture as presented in Figure 2.2, long displacements which commute between frames, etc. Similarly, the apparent motion can lead to a false correspondence. For such situations, it is possible using matching algorithms like tracking and correlation, although currently there is much controversy about the advantages and disadvantages of using these techniques rather than those based on gradient and energy of motion.

The correlation methods are less sensitive to changes in lighting and able to estimate long journeys that do not meet the sampling theorem [43]. However, they are extremely sensitive to periodic structures, providing multiple local minima and obtaining practically unpredictable responses when the aperture problem arises.

Alternatively, the other methods are better in efficiency and accuracy, and also able to estimate the optical flow perpendicular in the presence of the aperture problem. Moreover, the other methods behave better for short trips because of the size of local filters used.

Typically in machine vision, CCD (Charge Coupled Device) cameras are used with a discrete ratio, which varying modifies the displacement between frames. Consequently, gradient methods fail whether these shifts are too large, because they fracture the continuity of space time volume. Although it is possible to avoid temporal aliasing using an anti-aliasing spatial smoothing [44 – 45], the counterpart is to degrade spatial information. Therefore, for a given spatial resolution, it has to sample at a high temporal frequency [43].

On the other hand, it is quite common for real time optical flow algorithms to produce a functional architecture, as shown in Figure 2.4, via a hierarchical process. Firstly, it filters the sequence of images or the temporary buffer to extract basic measures through convolutions, FFT (Fast Fourier Transform), extraction of patterns, arithmetic, and so on. The measures are then recombined through various methods to reach a basic estimate of speed, usually incomplete and deficient in these early stages. Subsequently, the final estimate of flow is done by imposing a series of constraints on action and results. These are generated by assumptions about the nature of the flow or movement, such as restrictions of a rigid body. Even with these restrictions, the retrieved information is often not good enough to get a unique solution for optical flow field.

At the beginning of this section it was explained that the optical flow motion is estimated from the observable changes in the pattern of luminance over time. In the case of a no observable movement situation, such as a sphere rotating with the same brightness shown in Figure 2.2, the estimated optical flow is zero everywhere, even if current speeds are not zero. Another consideration is given by the no existence of a unique movement of the image, in order to explain a change in the observed brightness. Therefore, the visual motion measurement is often impossible and always has to be associated with a number of physical interpretations.

### *2.2.2. BMPs (Basic Movement Pattern)*

Despite the difficulties in the recovery of flow, biological systems work surprisingly well in real time. In the same way that these systems have specialized mechanisms to perceive color and stereopsis, there are also dedicated visual motion mechanisms [46]. As in other computer vision research areas, there are formed models of such natural systems to formalize bio-inspired solutions.

Thanks to psychophysical and neurophysiological studies, it has been possible to build models which extract the motion from a sequence of images. These biological models are usually characterized to be complex and designed to function poorly at high speed in real time.

One of the first models based on a real time bio-inspired visual sensor was proposed by Reichardt [47 – 48]. The detector consists on a pair of receptive fields as shown in Figure 2.5, where the first signal is delayed respect to the second before being nonlinearly combined by multiplication. Receptors 1 and 2, shown as edge detectors, are spaced a distance ΔS, imposing a delay on each signal and combining the result in C through multiplication. At the end, the result of the first half of the detector is subtracted from the second, and then it is estimated what each contributes to increased directional selectivity. The sensors shown in Figure 2.5 are Laplacian detectors, although it is possible to use any spatial filter or feature detector.

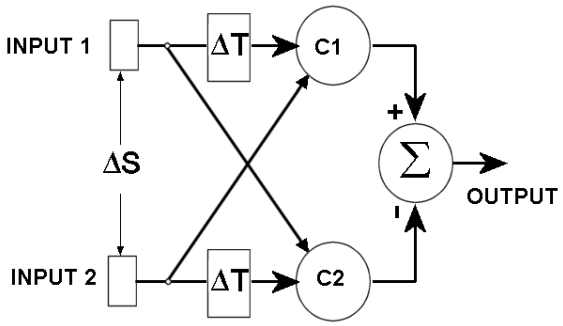


*Figure 2.5: Reichardt real-time correlation model [37].*

One of the main disadvantages of this detector is that the correlation is dependent on the contrast. In addition, no speed can be retrieved directly requiring banks of detectors calibrated at various speeds and directions, and its interpretation is ambiguous. However, despite these drawbacks, the Reichardt detector can be easily applied by biological systems and it is used successfully to explain the visual system of insects. The detector model continues being used as a starting point for more sophisticated models of vision [49 – 50], while detector hardware can be implemented in real time CCD sensors using VLSI (Very Large Scale Integration) technology [51].

- ***Change detection.***

If we consider the simple case of a segmented region moving relative to static regions, it appears a binary image indicating regions of motion, like other starting phases, for further analysis. The process may intuitively seem easy, so they are simply looking for changes in image intensity over a threshold, which is supposed to cause the movement of an object in the visual field. However, the number of false positives that

stem from different sources, such as noise sensors, camera movement, shadows, environmental effects (rain, reflections, etc.), occlusions and lighting changes, make very difficult to detect movement robustly.

Biological systems are also very robust to noise and uninteresting visual effects despite being very sensitive to movement. This technique is used in situations where motion detection is an event that should be taken into account for future use. Currently, estimation requirements for these algorithms are minimal, reaching a satisfactory result with little more than an input buffer, arithmetic signs, and some robust statistics. Moreover, as supervisory systems, they have to be very sensitive, and they are not normally available in large computing power [52 – 53].

- ***Correlation methods.***

When the differential approaches are subject to errors due to noncompliance with the sampling theorem [42], or inconvenience lighting changes, it is necessary to apply other strategies. Correlation methods or pattern matching are the most intuitive procedures able to regain speed and direction of movement, being their characteristic flow to select a frame in the sequence of images, and then look for these same characteristics in the next frame as shown in Figure 2.6. Hence, changes in the position indicate movement in time, i.e. speed.

These algorithms are characterized by a very slow performance due to their exhaustive search and iterative operation, requiring usually a prohibitive amount of resources. Being $M^2$ the image size, $N^2$ the search template size, and $L^2$ the search window size, the total estimation of required computational complexity would be around $M^2N^2L^2$. By way of example, with an image size of 512 x 512 points, and a template size and a search window size of 10 x 10, it would be required to run around 2.6 billion operations. The current trend is trying to reduce the search domain [37][55 – 56], although the needed resources are still too high.

One of the most common application models used is the encoding of video in real time [56 – 57]. Indeed, the amount of effort devoted to research in this type of algorithms is recently increasing.

*Figure 2.6: Block-Matching Technique [37].*

A video compression key observation is the similarity between temporarily adjacent images in a sequence. Hence, it requires less bandwidth to transmit the differences between frames than to transmit the entire sequence, being possible to reduce the volume of data transmitted even further if the movement and deformation needed to move from one frame to the next one are known a priori.

This type of algorithms can be preliminary classified in these two prominent approaches:

- Correlation as a function of 4 variables, depending on the position of the window and displacement, with normalized output between 0 and 1, being independent of lighting changes.

- Minimizing the distance, quantifying the dissimilarity between regions. Many optimizations take the approach of reducing the search space [37] and increase their speed [56].

Adelson and Bergen [58 – 59] advocate no biological evidence of such models, since they are not able to make predictions about complex stimuli, with which experimental observers perceive different moves in different positions, for example randomly positioned vertical bars. These techniques are simple, but many years have been spent researching, and dominating industrial inspection and quality control. One of their main advantages is the ability to work in environments where the displacements between frames are longer than a few points, though this requires processing longer search spaces.

- ***Space time methods: Gradient and Energy.***

Movement can be considered as an orientation in the space time diagram. For example, Figure 2.7 (left) presents a vertical bar continuously moving from left to right, sampled four times over time. Examining the space time volume, we can observe the movement of the bar over the time axis, and a stationary bar oriented parallel to that axis forming an angle, which will be showing the movement extent.

The orientation of the space time structure can be retrieved through low level filters, being two the currently dominant strategies, the gradient model and the model of energy, as shown in Figure 2.7 (right) where the ellipsis represents the negative and positive lobes.

The gradient model uses the ratio of a spatial and a temporal filter as a measure of speed, however the energy model uses a set of filter banks oriented in space time. Both models use a type of bio-inspired tightly matched filters [58 – 59][60].

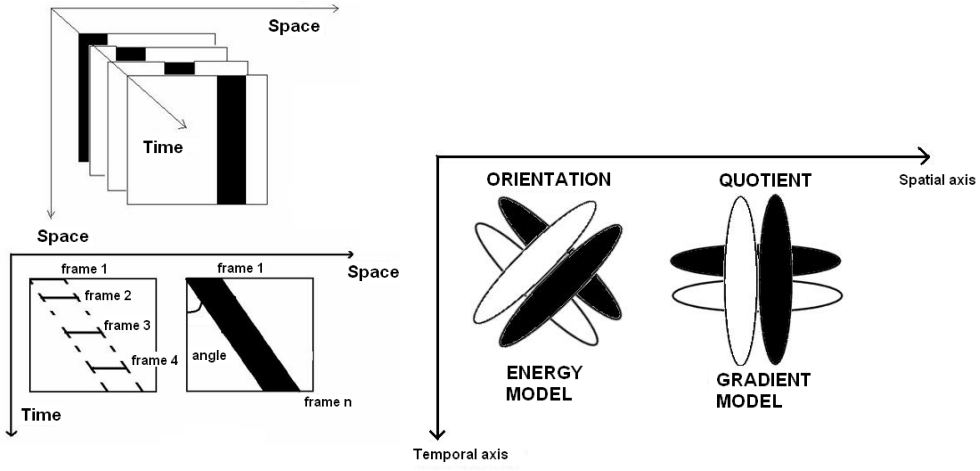


*Figure 2.7: Motion as orientation in the space-time (x – t plane) where the angle increases with the velocity (left). Space time filtering models (right).*

The controversy about which is the scheme adopted by the human visual system still remains open. Even, there are gateways to go from one model to the other, because it is possible to synthesize the filters oriented in the pattern of energy through space-time separable filters [41][61]. It is also interesting to note that ICA (Independent Component Analysis) says this type of spatial filters are those which cover the great majority of components of the image structure [62]. In the gradient model, the main

working hypothesis is the conservation of intensity over time [63 – 64], assuming this over short periods of time, the intensity variations are due only to translation and not to lighting changes, reflectance, etc. The total derivative of image intensity respect to time is zero at every point in space time, so defining the image intensity as $I(x, y, t)$, we have:

$$\frac{\partial I(x,y,t)}{\partial t} = 0 \tag{2.1}$$

Differentiating by parts we obtain the so called motion constraint equation:

$$\frac{\partial I}{\partial x}\frac{dx}{dt} + \frac{\partial I}{\partial y}\frac{dy}{dt} + \frac{\partial I}{\partial t}\frac{dt}{dt} = 0 \tag{2.2}$$

$$\frac{\partial I}{\partial x}u + \frac{\partial I}{\partial y}v + \frac{\partial I}{\partial t} = 0 \tag{2.3}$$

Where $u = \frac{dx}{dt}$ and $v = \frac{dy}{dt}$ and the parameters $(x, y, t)$ are omitted for the sake of clarity. Since there is only one equation with two unknowns, both unknown velocity components, it can be recovered only the velocity component $v_n$, which is in the direction of the gradient of luminance.

$$v_n = \frac{-\partial I/\partial t}{\sqrt{\left(\partial I/\partial x\right)^2 + \left(\partial I/\partial y\right)^2}} \tag{2.4}$$

There are several problems associated with the constraint equation of motion, because it is an equation with two unknowns, and therefore it is incomplete for estimating the optical flow. Using just one equation (2.3), it is only possible to obtain a linear combination of velocity components, being also fully consistent with the aperture problem mentioned before in the present section. A second problem arises whether $Ix$ or $Iy$, spatial gradients, become very small or zero, in which case the equation becomes ill-conditioned and estimated speed tends asymptotically to infinity. Furthermore, the stable realization of the spatial derivatives is something in itself problematic, applying a differential filter convolution, as operators of Sobel, Prewitt, or difference of Gaussians.

As they are using numerical derivatives of a sampled function, they are best suited for space time small intervals, although the problem of aliasing will appear every time the sampling in space-time is not enough, especially in the time domain as commented before. There are several filtering techniques to solve this problem, such as a spatiotemporal low-pass filtering as noted by Zhang [45].

Ideally, the sampling rate should be high enough to reduce all movements within one pixel/frame, so the temporal derivative is well-conditioned [42]. Moreover, the differential space-time filters which are used to implement gradient algorithms seem reasonably to those found in the visual cortex, although there is no consensus on the optimal from the functionality point of view [60]. One advantage of models on the energy gradient is that they provide a speed from a combination of filters, although energy models deliver a population of solutions.

In the framework of these models, we have to remark the optical flow error is commonly measured used the so-called Barron metric, [65] existing also others accepted by scientific community such as Galvin metric [66 – 67].

### *2.2.3. Improved motion constraint equation.*

We have seen that the MCE (Motion Constraint Equation) has several anomalies which have to be addressed properly to estimate optical flow. Indeed, there is a wide range of methods to improve it.

Many restrictions apply to resolve the two velocity components, $u$ and $v$, collecting more information through the acquisition of more images, or getting more information from each image, or alternatively applying physical restraints to generate additional MCEs:

- Applying multiple filters [30][68 – 70].
- Using a neighborhood integration [71 – 75].
- Using multispectral images [76].

Moreover, there are different general methods for restricting MCE and improve optical flow measures:

- Conducting global optimizations such as smoothing [63 – 64][77].

- Restricting the optical flow to a specific known model, for example the affine model [36][78 – 80].

- Using spatial and temporal multi-scale methods [55][81].

- Exploiting temporal consistency [82 – 83].

- ***Multiple Filters.***

As indicated previously, a motion constraint equation is not sufficient to estimate the optical flow by itself. An improvement of it has been proposed, taking into account the fact that its partial derivatives provide additional solutions to the flow vectors [84 – 85]. Nagel [84 – 85] was the pioneer in applying this method using second order differentiates; in fact, the differential operator is one of many that could be used to generate multiple MCEs. Usually these operators are used numerically by convolutions as linear operators.

This process works because the convolution does not change the orientation of the space-time structure. On the other hand, it is important to use filters which are linearly independent, otherwise the produced MCEs will degenerate and it will not have won anything. The filters and their differentials can be estimated in advance, to gain efficiency and, due to the locality of the operators, a massively parallel implementation of these structures.

- ***Multiple Integration***

It is possible using information stored on a local region to generate motion constraint equations extras [39][72][74 – 75]. Therefore, it is assumed that the movement is a pure translation in a local region, where these constraints are modeled using a weight matrix, so that the results are placed centered within a local region, as for example following a Gaussian distribution. Then, the MCE is rewritten as a minimization problem. The error term is minimized, or the set of equations generated by numerical methods are solved.

There are some especially robust models based on taking multiple derivatives in addition to multiple integration, which have been proved robust against noise and able to detect second order motion (McGM) [86 – 90].

- ***Multispectral methods.***

Working with multicolored images, different functions of brightness can be generated. For example, the planes of red, green, and blue of a standard camera can be treated as three separate images, producing three MCEs to solve. As a counterpart to this method, it should be noted that the color planes are usually correlated, which is exploited by most compression algorithms. In these situations, the linear system of equations can be degenerated, so that there is no guarantee that the extra cost in computing leads to an improvement in the quality of flow.

A variant of this method is to apply additional invariance with respect to small displacements and lighting changes, basing these measures in the ratio of different planes of color, spectral sensitivity functions, such as RGB (Red Green Blue) or HSV (Hue Saturation Value). Using this last variant, significant improvements are obtained over the use of a single RGB plane [76].

- ***Optimization.***

Due to the lack of information and spatial structure of the image, it is not easy to estimate a sufficiently dense velocity field. To correct this problem, several restrictions are applied, as for example, that the points move closer together in a similar way. The general philosophy is that the original flow field, once estimated, is iteratively regularized with respect to the smoothing restriction.

The first constraint was proposed by Horn and Schunk [63 – 64]. Optic flow resulting from the global constraints is quite robust, due to the combination of results, and also flattering to the human eye. Two of the biggest drawbacks are its iterative nature, requiring very large amounts of time and computing resources, and that motion discontinuities are not handled properly, so that erroneous results are produced in the regions surrounding the motion edges. To address these latter gaps other techniques which use global statistics such as random Markov chains are proposed [77].

- ***Movement patterns.***

In all estimation techniques there are serious restrictions on a neighborhood, where it is assumed constant flux. To meet this requirement, this neighborhood has to be as small as possible, but at the same time, it must be large enough to obtain information and to avoid the opening problem. Therefore, we need a trade-off compromise.

A variety of models use estimations related to this neighborhood, such as least squares. Using a quadratic objective function inherently assume Gaussian residual error rate, but if there are multiple movements in the neighborhood, these errors can no longer be considered Gaussian. Even if these errors were independent, which is very common, the error distribution could be modeled as bimodal.

There are approximate models which can be incorporated into the flow range of techniques that are being exposed. These approaches also model spatial variations of multiple movements. As mentioned, the neighborhood integration techniques assume that the image motion is purely translational in local regions, so more elaborate models such as the affine model can extend the range of motion and provide additional restrictions. These methods recast the MCE with an error function which will resolve or minimize least squares [82 – 83][91 – 94].

- ***Multiscale methods.***

Large displacements between frames originate gradient methods behave inappropriately, since the image sequences are insufficient or the time derivative measures are inaccurate. As a workaround, it is possible to use larger spatial filters [44].

The use of a multiscale Gaussian pyramid can treat large movements between frames and fill the gaps in large regions where the texture is uniform, so that estimations of coarse scale motion are used as sources for a finer scale [45].

The use of temporal multiscale [43] also allows the accurate estimation of a range of different movements, but this method requires using a sampling rate high enough to reduce movements about a pixel/frame.

- ***Temporal consistency.***

The schemes discussed so far, try the calculation of optical flow as a separate problem for each frame, without any feedback. Indeed, the motion results of a frame do not report the analysis to the following. Giaccone and Jones [82 – 83] have designed an architecture capable of dealing with multiple motions, and segmentation of moving regions by using a method of least squares.

This algorithm has proven very robust for a given speed range, and it also works well when compared to similar models. The cost calculation is overwhelmed by the generation of a projected image, PAL (Phase Alternating Line) sizes needed for about 40 seconds/image. However, this time consistency constraint is only used very sporadically today. The objects in the real world must obey physical laws of motion, such as inertia and gravity, so that there is predictability in their behavior, and it is at least surprising, that most real-time algorithms do not implement a flow based on feedback from previous instants.

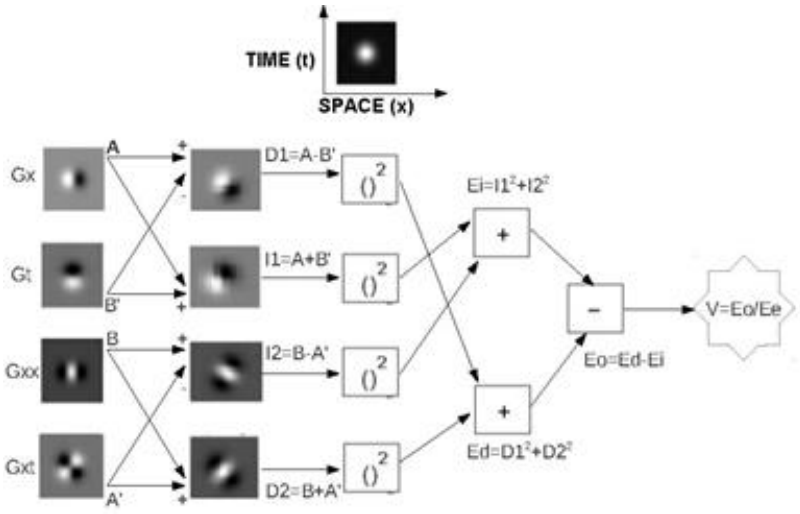


*Figure 2.8: Motion energy model.*

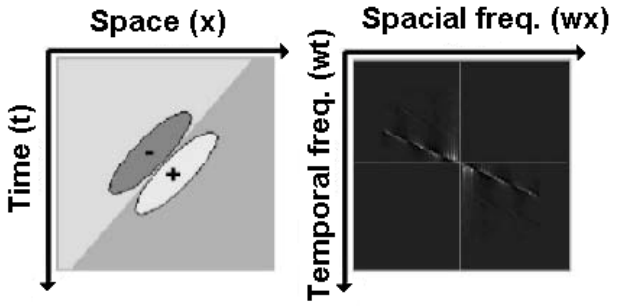


*Figure 2.9: Oriented filter in space-time which responds to the movement.*

For this purpose, it is possible to generate an additional constraint equation from the velocity field for using in the next iteration, managing the problem as an evolutionary phenomenon. The use of probabilistic or Bayesian models [75] may be an alternative, using real world information and updating results of previous estimations integrating temporal information.

### *2.2.4. Motion energy models.*

We have seen that the perception of motion can be modeled as an orientation in space-time, where the gradient methods extract this orientation across the oriented filter ratio. The motion energy models are often based on or are very similar in many aspects to gradient models, since both systems use filter banks to obtain this time-space orientation, and therefore the motion. The main difference is the filters used in energy models are designed to meet time-space directions rather than a ratio of filters. The design of space-time oriented filters is usually performed under the frequency domain.

Energy motion methods are biologically plausible, but their implementations have an extra high computational cost associated due to the large number of required filters, being their implementation difficult in real-time. The resultant velocity of energy methods is not obtained explicitly, unlike gradient methods, using only a solution population, being these last Bayesian models.

One advantage is that the bimodal velocity measurements, as in invisible movements, can be treated by these structures [75]. The correct interpretation of the processed results is not an easy task when dealing with models of probabilistic nature. Interesting

optimizations have been developed to increase the speed of these methods, combined with Reichardt detectors [95] as support.

## 2.3. Accelerators implemented for motion estimation.

Techniques for estimating optical flow often seek a compromise between accuracy and efficiency [96]. This compensation arises because the most precise techniques tend to have higher computational requirements. Given similar computational resources, some techniques make more precise estimations but slower, while others get less accurate movement calculations but faster. The accuracy allows assessing the quality of the results obtained while the efficiency refers to both, the time in which the data are obtained, as to the computational resources used for it.

In motion estimation, particularly in the context of this thesis, it is interesting to obtain efficiently accurate results in real-time, using techniques that allow their adaptation to real problems. To this aim, we will do a review of studies in the past years related to motion estimation accelerators, including those which make use of graphics hardware as specific circuits, and FPGAs.

### *2.3.1. FPGAs (Field Programmable Gate Array).*

One of the types of accelerators which are used for motion estimation is the FPGA. Due to this reason, multiple related studies are found in the literature. This architecture has been commented in Chapter 1 and will be addressed in the framework of embedded systems at Chapter 4. Table 2.1 presents some of those made in the last years, showing also the implemented algorithms family, the performance obtained, and the type of FPGA used.

| Author | Family | Algorithm | fps (resolution) | FPGA | Year |
|---|---|---|---|---|---|
| Niitsuma [97] | Matching | FST | 30 (640×480) | Virtex-II XC2V6000 [98] | 2004 |
| Babionitakis [99] | Matching | Many | 30 (1024×768) | Virtex-II Pro 40 | 2008 |
| Asano [100] | Matching | FST | 30 (1920×1080) | Virtex-5 [101] | 2010 |

| | | | | | |
|---|---|---|---|---|---|
| Akin [102] | Matching | OBT [103] | 83 (1920×1080) | NP | 2010 |
| Ho [104] | Matching | MFHME [105] | 30 (1920×1080) | NP | 2011 |
| Núñez-Yáñez [106] | Matching | FME [107] | 62 (1920×1080) | Virtex-5 | 2012 |
| Wei [108] | Gradient | Tensors [109 – 110] | 64 (640×480) | Virtex-II Pro XC2VP30 [98] | 2007 |
| Díaz [111] | Gradient | Lucas & Kanade [71] | 170 (800×600) | Virtex-II XC2V6000-4 | 2008 |
| Botella [112] | Gradient | McGM [86 – 87] | 177 (128×96) | Virtex-E 2000eBG560 [113] | 2010 |
| Bahar [114] | Gradient | Horn & Schunck[64] | 1029.9 (240×320) | Cyclone II [115] | 2012 |
| Barranco [116] | Gradient | Lucas & Kanade[71] | 270 (640×480) | Virtex-4 XC4vfx100 [117] | 2012 |
| Gultekin[118] | Gradient | Horn & Schunck[64] | 257 (256×256) | EP2C70 Cyclone-2 | 2013 |
| Monson[181] | Gradient | Lucas & Kanade[71] | 42 (NP) | Xilinx Zynq-7000 | 2013 |
| Komorkiewicz[127] | Gradient | Horn & Schunck[64] | 60 (1920×320) | Virtex -7- XC7VX980T | 2014 |
| Norouznezhad [120] | Energy | Gabor filter [121] | 30 (40×480) | Xilinx Virtex-5 XC5VSX50T | 2010 |
| Tomasi [122] | Energy | Phase-Based [123] | 36 (512×512) | Virtex-4 XC4vfx100 | 2010 |
| Tomasi [124] | Energy | Phase-Based [125] | 22.8 (512×512) | Virtex-4 XC4vfx100 | 2012 |
| Paul [126] | Energy | Tunley[118] | 31 (NP) | Virtex-5 | 2013 |

*Table 2.1: State of the art motion estimation algorithms under FPGAs. "NP" means Not Provided.*

Niitsuma and Maruyama [97] implemented the FST (Full Search Technique) exhaustive search algorithm in an FPGA in order to calculate the motion of objects and the distance at which these are found, by combining the optical flow in stereo vision. The performance achieved by this system is 30 fps (frames per second) for a resolution

of 640×480. Other work related to motion estimation by block-matching using FPGAs is disclosed by Babionitakis et al. in [99], where an architecture that efficiently supports a set of block-matching algorithms is presented. In addition, the proposed design runs the different algorithms by providing a set of instructions common to all these algorithms, and only a few specific instructions to each of them. The results obtained on an FPGA are 30 frames per second with a frame size of 1024×768.

In motion estimation using the block-matching paradigm, the current frame is divided into macroblocks to find the best matching block for each macroblock in the reference frame of the search. However, Asano et al. [100] proposed a method by which the macroblock exploration direction in the current frame and the coincidence exploration direction in the search area are optimized in order to reduce external memory banks access, which store the reference frame, caching in memory the search area representation. By reducing both memory accesses, it is possible to obtain high performance in a FPGA getting 30 fps for a resolution of 1920×1080. Consequently, it also reduces the number of memory banks required to process this data in real-time. To do this, they show an approximation for motion estimation by means of a complete search with a VSBME (Variable Size Block Motion Estimation) on that device.

Considering FPGA implementations of matching models, we remark the work done by Akin et al. [102], concerning high performance hardware architectures for algorithms based on 1BT (One Bit Transform), where several models of architectures that are able to obtain faster results using less memory estimation are discussed. Furthermore, they are the first proposing a reconfigurable hardware for motion estimation based on multiple frames, by which the number and selection of reference frames can be configured in a static way basing on the requirements of the application, in order to estimate the performance and computational complexity of the motion. The performance achieved in this work is 83 fps using a frame resolution 1920×1080. Meanwhile, Ho et al. [104] present an implementation in FPGA of MFHME (Multi Frame Hierarchical Motion Estimation) algorithm, which manage to reach 62 fps for conversion of high definition video (1920×1080). The scalable system is thoughtfully designed, and gets high performance video in real time with high precision.

Concluding works concerning matching models algorithms, Nuñez-Yañez et al. [106] present a flexible and scalable implementation for motion estimation which is able to support the requirements for processing high definition video using H.264 codec AVC3.0[10]. To do this, the algorithm optimizes fast block-matching obtaining a yield of 62 fps for resolutions of 1920×1080.

About works related to gradient models, we will highlight the contribution made by Wei et al. [108], which explains how it has been implemented and modified an optical flow algorithm based on tensor, to adapt it to a FPGA in order to avoid obstacles and enable navigation of unmanned vehicles everywhere. The yield obtained is 64 fps for a frame size of 640×480, and an average angular error of 12.9º (Barron's metric) is also achieved.

Regarding the Lucas & Kanade algorithm under FPGAs, the work carried out by Dıaz et al. [111] is the implementation of the Lucas & Kanade algorithm in FPGA with the ability to process up to 170 fps at a resolution of 800×600 pixels on a scalable, modular, and versatile architecture. Furthermore, Barranco et al. [116] proposed a parallel architecture for emotion estimation that is capable of achieving 270 fps for a resolution of 640×480 in the best case of a monoscale implementation, and 32 fps for a multiscale implementation [128 – 129] obtaining an average angular error of 4.55º in the best case.

Regarding the implementation of the McGM (Multichannel Gradient Model), Botella et al. showed it at work [112], based on customizable reconfigurable hardware architecture with the properties of the sequence of crustal movement. The performance obtained is 177 fps with a resolution equal to 128×96 and an average angular error of 7.2º. The approach shown in this paper is appropriate in situations in which the luminosity is highly variable, in noisy environments, or in research on the perceptual system of humans.

Another algorithm implemented in FPGAs belonging to gradient models is presented in the work of Bahar and Karimian [114], which exemplified the use of the Cyclone II FPGA to accelerate the calculations concerning to optical flow algorithm H&S (Horn &

[10] Standard for video compression: http://www.itu.int/rec/T-REC-H.264.

Schunck) [118]. The proposed implementation uses parallel storage units for accessing memory and calculations easily, improving the efficiency of the algorithm and then significantly reducing the clock cycles needed to run it, down to only one cycle. Thus, it is reached a maximum yield of 1029.9 fps for a resolution of 240×320, suitable for motion detection in real-time, with an average angular error of 6.96º.

Additionally, we present the work of Gultekin et al. [119], based again on the well-known H&S algorithm, optimized and implemented in FPGAs belonging to gradient models. They use a low cost board (DE2-70) with an FPGA from Altera (EP2C70 Cyclone-2). They are able to reach a rate of 257 fps at 256×256 resolution. They claim to obtain an average error < 1º for the famous Rubik sequence using the Barron's metric.

We also present some of the works on FPGAs under the framework of optical flow energy based paradigm. Norouznezhad et al. [120] present a system implemented in FPGA for object tracking using Gabor filters through various channels of the input image. The system is able to obtain a yield of 30 fps input for a resolution of 640×480.

Other model is presented by Monson et al. [119] regarding "C" code running on a FPGA platform. In this paper, "C" code for a Lucas & Kanade optical flow algorithm is optimized for both a desktop PC (Personal Computer) and a FPGA based system, the Xilinx Zynq 7000, which contains both, a programmable fabric and two ARM cores. Their paper discusses how the code is optimized and restructured to be executed effectively on the programmable fabric and the two ARM cores. The resulting system under the Xilinx Zynq 7000 is competitive with the desktop PC but only consumes 1/7th as much energy.

Additionally, Paul et al. [126] use the energy based algorithm of Tunley [118] to compensate and estimate ego motion, allowing object detection from a continuously moving robot using a first order motion model. This system is able to deliver 25fps (frame size is not provided) using a Virtex 5 FPGA. Also, this algorithm is theoretically able to deal with strong rotation and translation in 3D, with 4 degrees of freedom.

Another example is disclosed by Tomasi et al. [122][124], having two papers where phase based algorithms are implemented in FPGA. An architecture based on a

multiscale which is capable of processing images with a resolution of 512×512 at 36 fps accurately is implemented through a phase model at [122]. The algorithm calculates the phase for eight different orientations on a pyramid model, and propagates the information through an adequate number of scales as a size function of input image and filter size. Additionally in [124] is given an architecture for the extraction of low-level vision characteristics including the multiscale optical flow, the disparity, the energy, the orientation, and the phase. From the presented solutions, the high performance computing is capable of 165 GOPS with a power consumption of 30 GOPS/W using a clock frequency of 50 MHz achieving a system performance of 22.8 fps with a frame size of 512x512.

Additionally, Komorkiewicz et al. [127] present an efficient hardware implementation of the H&S algorithm that can be used in an embedded optical flow sensor. An architecture which realizes the iterative H&S algorithm in a pipelined manner is proposed. This modification allows achieving data throughput of 175 MPixels/s and processing Full HD video stream (1920×1080 @ 60 fps) possible. The described architecture was tested on popular sequences from an optical flow dataset of the Middlebury University. It achieves state of the art results among hardware implementations of single scale methods. Also, it is presented a complete working vision system realized on the Xilinx VC707 evaluation board. It is able to compute optical flow Full HD video stream received from an HDMI camera in real-time. The obtained results prove that FPGA devices are an ideal platform for embedded vision systems.

### *2.3.2. GPUs (Graphic Processing Unit).*

GPUs are another recent development based on a customized processor primarily for graphics rendering, initially designed for entertainment market. Nowadays, current GPUs have a large number of processors which can be used for general purpose computing. Supercomputers that currently lead the world ranking combine the use of a large number of CPUs with a high number of GPUs. The GPU is especially appropriate to solve computationally intensive problems which can be expressed as data parallel computations. However, implementation on GPU requires the redesign of the algorithms, focused and adapted to its architecture taking into account two key

characteristics: the complex memory hierarchy optimization used, and the efficient mapping of the problem to be parallelized into block of threads, and threads themselves.

Programming GPUs automatically involves considering a number of constraints, such as the need for high occupancy in each processor in order to hide latencies produced by management, the synchronization of different threads running simultaneously, memory access, the proper use of the hierarchy of memories as remarked, and other considerations. Researchers have already successfully applied GPU computing to problems which were traditionally addressed by CPU.

We present in Table 2.2 an overview of the last and more representative works regarding acceleration of motion estimation using GPUs. The GPU implementation used in these works in mainly based on NVIDIA's CUDA (Compute Unified Device Architecture) architecture, which is supported by most current NVIDIA graphics chips. Despite of it, OpenCL (Open Computing Language) is becoming and standard that will unify many architectures such as Multicore, DSP (Digital Signal Processor), GPU and FPGA among others.

| **Author** | **Family** | **Algorithm** | **fps (resolution)** | **FPGA** | **Year** |
|---|---|---|---|---|---|
| Kiss [130] | Matching | Crosby [131] | 33 (200×200) | GTX 285 [132] | 2009 |
| Zhang [133] | Matching | BNM [134] | 15 (1280×768) | Tesla C1060 [135] | 2010 |
| Monteiro [136] | Matching | FST | 256 (1280×768) | GTX 480 [137] | 2011 |
| Ranft [138] | Matching | Raw Block-Matching [139] | NP | GTX 470 [140] | 2011 |
| Zhang [141] | Matching | FST | 89 (1280×720) | Radeon HD 6870 [142] | 2012 |
| Rodríguez-Sánchez [143] | Matching | NP | 6 (1920×1080) | GTX 480 | 2012 |
| Monteiro [144] | Matching | FST<br>DS | 33 (1280×720)<br>310(1280×720) | GTX 480 | 2012 |
| Vu [145] | Matching | HS [146] | 16 (1280×720) | Tesla C2050 [147] | 2012 |

| | | | | | |
|---|---|---|---|---|---|
| Chase [148] | Gradient | Tensor [108] | 150 (640×480) | 8800 GTX [149] | 2008 |
| Marzat [150] | Gradient | Lucas & Kanade | 47 (316×252) | Tesla C870 [135] | 2009 |
| Duvenhage [151] | Gradient | Lucas & Kanade | 30 (512×512) | GTX285 [132] | 2010 |
| Phull [152] | Gradient | RLCT [153] | 465 (1024×768) | 280 GTX [154] | 2010 |
| del Riego [155] | Gradient | HLK [128] | 30 (640×480) | 9500 GT [156] | 2011 |
| Hegner [157] | Gradient | Pyramid Algorithm [158] | 36 (512×512) | GTX 260 [159] | 2011 |
| Ohmura [160] | Gradient | Lucas& Kanade | 40 (512×384) | Tesla C1060 [135] | 2011 |
| Shiralkar [161] | Gradient | SOM [162] | 20 (640×480) | GTX480 [137] | 2012 |
| Ayuso [163] | Gradient | McGM [86 – 87] | 32 (256x256) | Tesla C2070 | 2013 |
| García [164] | Gradient | McGM [86 – 87] | 80 (256x256) | Tesla C2070<br>Tesla C1060 | 2013 |
| Pauwels [165] | Energy | Phase-Based [123] | 128 (316×252) | 8800 GTX [149] | 2008 |
| Sundaram [166] | Energy | LDOF [167] | 1.84 (640×480) | GTX 480 [137] | 2010 |
| Gwosdek [168] | Energy | Euler-Lagrange | NP | GTX 480 [137] | 2012 |
| Abramov [169] | Energy | Phase-Based [170] | 4.3 (640×512) | GT 240M [171] | 2012 |

*Table 2.2: State of the art motion estimation algorithms under GPUs. "NP" means Not Provided.*

Kiss et al. presented in [130] a way to optimize the performance of block-matching techniques for 3D eco cardiograms using a SIMD (Single Instruction Multiple Data) model. This compares an implementation of the algorithm developed in [131], using SSE (Streaming SIMD Extensions) instructions with an implementation on GPU using CUDA, getting the second implementation 26 times higher performance (26×) than the

first. In both cases, the time at which the algorithm is performed is reduced, which is an improvement when applied in clinical settings.

In the work of Zhang et al. [133] it is presented an implementation on GPU of BNM (Best Neighborhood Matching) algorithm [134], an effective method for recovering degraded images by poor transmission in the network, applied to images with high color definition using resolutions higher than 1280×768. The yield obtained is 21× with respect to a sequential version of the same algorithm running on a CPU.

The study conducted by Ranft et al. [138] focuses on three different hardware platforms, a multicore x86 system, a GPU, and a 64 core embedded system, to parallelize matching estimations, reaching the conclusion that the GPU is the fastest of the three, achieving between 1.3× and 4.3× improvement.

With respect to the FST algorithm, Monteiro et al. [136] present an implementation of this algorithm using CUDA in order to get high performance when estimating motion in video coding. In their paper, the obtained performance is studied by comparing it to an implementation that uses sequential OpenMP[11], and to another running on the CPU, giving a yield of 66× and 600× respectively. Furthermore, the work in [144] presents efficient algorithms to map motion estimation on GPU using the CUDA programming model method. The chosen algorithms for the study were FST and DS (Diamond Search). To compare the results, it is also implemented a version of the algorithms using OpenMP and MPI33, in order to have a version under multicore and distributed respectively. The solution provided using CUDA gets better performance than the versions under OpenMP and MPI. Thus, for the FST algorithm, the performance obtained with respect to a sequential version is 154× for the CUDA implementation, while 13× with MPI, and 1× under OpenMP. In addition, comparing the version of CUDA respect to MPI[12] and OpenMP versions, speedups of 14× and 77× respectively are obtained. Regarding DS algorithm, speedups of 62×, 1×, and 0.5×, for CUDA, MPI, and OpenMP versions are obtained respectively. In addition, the CUDA version achieves a speedup of 77× with respect to the MPI version and 191× with respect to the OpenMP version.

[11] OpenMP: http://openmp.org/wp/.

[12] MPI(Message Passing Interface): http://www.mcs.anl.gov/research/projects/mpi/.

Zhang Jinglin, and Cousin Nezan introduced in [141] an implementation based on motion estimation with heterogeneous parallel computation. Using OpenCL[13] it is proposed a method to determine the distribution of workload in a heterogeneous system with a CPU and a GPU, achieving speedups between 100× and 150× compared to the same implementation in one CPU. For testing, it is compared OpenCL implementations on CPU, NVIDIA card, and another ATI card. This last is where they get the best results with 89 fps at resolutions of 1280×720. Also, it is interesting to note that the criteria for selecting the best motion vector are computed by the SAD algorithm.

In [143], Rodríguez-Sánchez et al. show a method for encoding 3D high definition movement video using GPU, which reduces the execution time by up to 98% besides being energetically more efficient requiring less energy than the sequential method. This method is an extension of H.264/AVC codec to support 3D video, MVC (Multiview Video Coding) [172]. The results obtained in this study show a 67× speedup for the algorithm proposed, and 10× for the full video encoding.

To end with block-matching algorithms under GPU, the work of Vu, Yang and Bhuyan [145] present a dynamic and efficient approximation for GPU based HS (Hierarchical Search) algorithm [146]. To do this, a selection scheme, fixed and dynamic, is implemented obtaining a speedup equal to 200× and 250× respectively.

Regarding studies of gradient family algorithms under GPU, Chase et al. [148] present a comparison between the use of an FPGA (Virtex -II Pro Xilinx XC2VP30) and a GPU (NVIDIA GeForce 8800 GTX ) to calculate the optical flow. They reached the conclusion that the implementation on the FPGA requires more development time, about 12×. To do this, they implement a version of the algorithm described in [108] on both platforms. Under the GPU 150 fps are achieved with a frame size 640×480, while the FPGA achieves 64 fps for the same resolution. Meanwhile, Marzat et al. [150] implemented a pyramidal version of Lucas & Kanade algorithm using CUDA to accelerate it. They get a throughput of 100× in comparison with the counterpart CPU version, besides getting 15 fps for a resolution of 640×480.

[13] OpenCL (Open Computing Language): http://www.khronos.org/opencl/.

In their work, Duvenhage et al. [151], present a GPU implementation of the Lucas & Kanade algorithm in order to be used in image stabilization applications. To perform this task, they implement the algorithm using GLSL[14], reaching 30 fps for a size frame of 512×512 with the best settings.

Phull et al. [152] developed using CUDA the algorithm described in [153], where optimizing and accelerating the algorithm obtained speedups of 25× with respect to the implementation on CPU, and 236× considering the implementation on CPU of the pyramidal KLT (Kanade Lucas Tomasi) algorithm [173].

Nowadays, there are many devices where we can find a GPU to handle 3D graphics and other related tasks, like for example embedded cameras and laptops, or game consoles and mobile phones between others. Using the camera duo + GPU, Irrigaton et al. [155] implemented in a graphics processor a parallel version of the optical flow algorithm HLK (Hierarchical Lucas Kanade) [128]. In this implementation, they obtained 30 fps, the maximum supported by the camera used in their experiments, besides getting reduce the percentage of CPU usage, since the calculations are performed on GPU.

The work developed by Hegner et al. [157] presents the implementation on a GPU of an optical flow algorithm developed in [158] using directional signals filters. With this, the obtained results, for a sequence of input images of size 240×256, are thousands of times faster than the corresponding implementation in MATLAB (http://www.mathworks.com/products/matlab/index.html). The number of fps varies depending on the resolutions used as input data, achieving 148 fps for resolutions of 240×256 and 36 fps for 512×512.

There are two works regarding the McGM (Multichannel Gradient Model) algorithm belonging to the gradient family. The scheme was particularized using a validated neuromorphic motion estimation system for the robust extraction of image velocity. This model contains many characteristics that enhanced the capability when compared with other optical flow gradient family algorithms.

[14] OpenGL Shading Language: http://www.opengl.org/documentation/glsl/.

Ayuso et al. [163] describe an optimized implementation over a Tesla board achieving a 100% dense implementation with a throughput of 32× for a 256×256 frame resolution releasing 185 fps.

Garcia et al. [174] implemented a multi GPU version which is able to tune the McGM algorithm in real-time using a NSGA-II optimizer. This last system is designed to overcome the problem of the memory consumption which creates a bottleneck due to the expansive tree of signal processing operations performed. In this contribution, an improvement in memory reduction was carried out, which limited accelerator viability usage. An evolutionary algorithm was used to find the best configuration. It supposes a trade-off solution between consumption resources, parallel efficiency, and accuracy. A multilevel parallel scheme was exploited, where the grain level by means of multi GPU systems, and a finer level by data parallelism. The system is able to deliver real-time results.

The numerical simulation for visual processing of the human brain is one of the most time consuming applications. In this line, Ohmura and his team [160] show techniques to accelerate a program that simulates performing visual processing motion estimation by Lucas & Kanade algorithm. To do this, the convolution parallelized is loaded on a cluster of GPU Tesla C1060 and C2070 [135]. This research includes several improvements such as efficient allocation of data between global memory and shared memory of the GPU. Each GPU calculate multiple convolutions for the same input data and thanks to the division into regions of the input image, these regions are executed in parallel using MPI (Message Passing Interface).

Meanwhile, Shiralkar et al. [161] present a parallel version of the SOM (Self Organizing Map) algorithm [162]. In this work, the obtained performance is 140 fps using a 320×240 frame size, while for 512×284 and 640×480 sizes it is obtained 40 fps and 20 fps respectively. In addition, depending on the input stimulus, reach speedups are between 130× and 145×.

To finish the work related to motion estimation under GPUs, we review some works based on energy models algorithms. Pauwels and Van Hulle proposed in [165] the use of a GPU architecture which implements using CUDA an optical flow algorithm based on phase [123]. The GPU implementation gets about 40 fps for resolutions of 640×512

and 150× speedup. This optical flow algorithm focuses on a measure of reliability that evaluates the consistency of the phase information in time, and is characterized by its simplicity, robustness and speed, but does not include stabilization measures, which is a crucial aspect when taking into account higher resolutions.

In [166], Sundaram et al. propose a method to calculate the trajectory of points by means LDOF (Large Displacement Optical Flow) algorithm [167]. This implementation runs 78× faster than the corresponding C++ version. Gwosdek et al. [168] present an implementation on GPU for computing the optical flow in the Euler-Lagrange system [64][175]. In this work it has been used the algorithm proposed in [176] to minimize the optical flow model approach, obtaining optimum results in less than a second to sequences sized of 640×480. For higher resolutions (1024×768), the speedup achieved varies between 23× and 28× depending on the input stimulus used.

Finally, the study by Abramov et al. [169] is based on a model of spatiotemporal segmentation in mobile robotic applications. To perform this work, it is mapped to GPU an algorithm based on phase [170] comparing the results obtained on a GPU Nvidia GeForce GT 240M installed on a laptop, and on a GPU Nvidia GeForce GTX 295 installed on a desktop[15]. Although the obtained results with the first card are of the order of 2.1× slower than those obtained with the second, it is still possible to render multiple frames per second for the resolutions considered in this study (160×128, 320×256 and 640×512).

### *2.3.3. Embedded microprocessors.*

In this subsection some works related to motion estimation using embedded microprocessors are shown, as can be seen in Table 2.3.

Processor manufacturers are now concerned about concepts such as green computing. The aim is to develop more efficient chips, not only in terms of performance rates, throughput measured in terms of FLOPS (FLoating point Operations Per Second) or Mbits per second, but also in energy efficiency [177].

[15] GeForce GTX 295 specifications: http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-295/specifications.

Besides modern and efficient multicore CPUs, hardware accelerators such as GPUs, Intel (Intel Corporation; Santa Clara, CA, USA) MIC (Many Integrated Core), or reconfigurable devices (FPGAs), one of the latest additions on specific purpose architectures applied to general purpose computing are low power DSPs (Digital Signal Processor).

One of the primary examples in this field is the C6678 multicore DSP from TI (Texas Instruments; Dallas, TX, USA) which combines a theoretical peak performance of 128 GFLOPS (Giga (billions) FLoating point Operations Per Second) with a power consumption of roughly 10 W per chip. Moreover, one of the most appealing features is the ease of programming, adopting well known programming models for sequential and parallel implementations.

| **Author** | **Family** | **Algorithm** | **fps (resolution)** | **FPGA** | **Year** |
|---|---|---|---|---|---|
| Hui [182] | Matching | FST/DCT | 42 (352x288) | TMS320C6455DSP | 2013 |
| Hong[184] | Matching | CST [184] | NP | ARM9 | 2011 |
| Honegger [186] | Matching | Matching FST | 127 (376x240) | Nios II | 2012 |
| Steiner [188] | Matching | Matching | | PIC33FJ128MC804 | 2011 |
| Igual [174] | Gradient | McGM [86 – 87] | 136 (64X64))<br>35 (64X64))<br>57 (64X64))<br>6 (64X64))<br>7 (64X64)) | Xeon (8c)<br>Xeon (1)<br>C6678 DSP<br>Atom<br>Cortex A9 | 2013 |
| Monson[181] | Gradient | Lucas & Kanade[71] | 42 (NP) | ARM and Xilinx Zynq-7000 | 2013 |
| Guzman[185] | Gradient | Lucas & Kanade[71] | 62 (176x144) | Nios II | 2010 |
| Anguita[180] | Gradient | Lucas & Kanade[71] | 68 (1280x1016) | Core 2 Quad PC | 2009 |
| Rowenkap [187] | Gradient | Horn & Schunck[64] | 15 (128x128) | TMS320C40 DSP | 1998 |

| | | | | | |
|---|---|---|---|---|---|
| Shiraii [189] | Gradient | Horn & Schunck[64] | NP | TMS320C40 DSP | 1990 |
| Lalonde[183] | | | 25 (NP) | 2xARM926 | 2012 |

*Table 2.3: State of the art motion estimation algorithms under embedded microprocessors. "NP" means Not Provided.*

The limitation in power consumption of current embedded devices makes necessary to consider energy related issues in the implementation of optical flow algorithms. There are in the literature ad hoc solutions to solve the motion estimation problem with power constraints. As an example, there are countless proposals under low power conditions for pattern matching family algorithms, but most are in the video compression field [178 – 179]. Another approach with CPUs (Central Processing Unit) [180] presents a parallel scheme applied to a model based on the well- known Lucas & Kanade approach, which reduces power consumption in terms of TDP (Thermal Design Power) and still meets the real-time requirements when low power chipsets (TDPs of 20 to 30 W) are used. Moreover, Honegger et al. [186] implement a low power stereo vision system under FPGA based on the Nios II processor (Altera, San Jose, CA, USA).

Igual et al. [174], provides an efficient implementation for an optical flow gradient based model using a low power DSP exploiting different levels of parallelism. To the best knowledge of the authors, this is the first attempt to use a DSP architecture to implement a robust optical flow gradient based model providing results in real-time.

There are only few approaches existing in the literature exploiting gradient based motion estimation methods in DSP platforms as the one proposed by Shirai et al. [189] in early 1990s, implementing the classical method of Horn & Schunck algorithm [190] using many boards where each have a TMS320C40 DSP.

This algorithm supplements optical flow constraint regularizing smooth terms, while Igual work [174] uses spatiotemporal constancy. Besides, performance and/or energy consumption is not considered in that work. Rowenkap et al. [187] implemented in 1997 the same algorithm as the previous work, using the same DSP and reaching a throughput of 5 fps or 15 fps, for 128×128 image resolutions, when using one or three DSPs respectively. The last work considered is the neuromorphic implementation of

Steiner [188] that uses the Srinivasan algorithm [191] on a PIC33FJ128MC804 DSP processor. This algorithm is based on simple stage procedure of image interpolation.

Hui et al. [182] implement a MPEG-4 video encoder and consequently a block-matching to satisfy requirements of the public transport's monitoring system. But, there exist many problems. These problems are that the transmission of each macroblock cache is seriously deficient, and the video frame rate sequence is low. The optimization of motion estimation algorithm, the DCT (Discrete Cosine Transform) optimization, the optimization of SAD, the optimization of pixel interpolation, the increase of the pre-zero DCT coefficient of the whole identification algorithm, and the optimization of the MPEG-4 video encoder on a TMS320C6455 DSP platform, are put forward in their paper with a throughput of 42 fps using a frame size of 352×288.

Monson et al. [181] present the implementation of the Lucas & Kanade algorithm. In their paper, "C" code for a complex optical flow algorithm is optimized for both, a desktop PC and a FPGA based system, the Xilinx Zynq-7000, which is a device containing both, a programmable fabric and two ARM cores. This paper discusses how the code is optimized and restructured to be executed effectively on the programmable fabric and the two ARM cores. The resulting system under the Xilinx Zynq 7000 is competitive with the desktop PC but only consumes 1/7th as much energy, releasing 10 fps with a resolution of 720×480.

Hong et al. [184] implements a low cost embedded system based on the ARM9 processor which uses a self-tuned CSM (Cross Searching Mode) algorithm belonging to the matching family. Unfortunately, they are not provided performance results in this paper.

Lalonde et al. [183] present an implementation of a 3D reconstruction algorithm for the detection of static obstacles from a single rear view parking camera. The authors adopted a feature based approach in which interest points are tracked to estimate the vehicle motion and multiview triangulation is performed to reconstruct the scene. A full implementation of the algorithm has been achieved on a parallel SIMD array processor unit embedded in a smart automotive camera system. Currently, in vehicle beta trials suggest that system performance meets industrial requirements for real world use in

backup camera systems. Only few implementation details are presented in this work, which achieves 25 fps.

Finally, Guzman et al. [185] present a motion sensor using a platform that includes the sensor itself, focal plane processing resources, and co-processing resources on a general purpose embedded processor. They make use of a commercial smart camera designed by AnaFocus, named the Eye-RIS™ v1.2 [192], with image resolution of 176×144 pixels and capable to operate above 10,000 fps. All this system is implemented on a single device as a SoC (System On a Chip). The authors show an approach to estimate the motion velocity vectors with an architecture based on a focal plane processor combined on-chip with a 32 bits Nios II processor [273]. This system is fully functional and organized in different stages, where the early processing stage, focal plane, is mainly focused to preprocess the input image stream to reduce the computational cost in the post processing stage (Nios II). The final outcome is a low cost smart sensor for optical flow computation with real-time performance and reduced power consumption that can be used for very diverse application domains.

### *2.3.4. ASICs (Application Specific Integrated Circuit).*

In comparison to dedicated hardware like a ASIC, a FPGA implementation is slower, requires more silicon, and more power. A custom made circuit will be smaller than a generic circuit for a number of reasons. The first reason is the programmable hardware requires extra configuration logic, which also consumes a significant proportion of the chip real estate. The second reason is the custom circuit only needs the gates that are required for the application, whereas the programmable circuit will have components that are not used. And the third reason is that a programmable interconnect is not required in a custom circuit, so the logic is just wired where it is needed. In a FPGA the interconnect logic must also be sufficiently flexible to allow a wide range of designs, so again it will only be partially used in any particular configuration.

Considering all reasons exposed above, we see that a usual FPGA requires 20 – 40 times the silicon area of an equivalent ASIC [193]. Considering speed, an ASIC will be faster than programmable logic because the wiring delays can be minimized connecting components as close as possible to each other. Moreover, due to the fact that FPGA circuits are bigger, they will reduce the maximum clock speed reached and will

consume 10 – 15 times more dynamic power since it has more capacitance and more transistors to be switched. Using FPGAs the logic blocks are spaced further, so signals take longer when travelling from one block to another, and this effect is getting worse when using tools for automated placing and routing. Besides, every connection which switches a signal introduces a delay. As consequence of the above considerations we find an ASIC typically 3 to 4 times faster than an FPGA [193] for the same level of technology.

The mask and design costs of ASICs are significantly higher although the cost per chip is significantly lower. This makes the ASIC only economical where the speed cannot be matched by a FPGA or high volumes are required. With each new generation technology, the capabilities of FPGAs increase, enabling a shorter time to market. because of both, the re-configurability allows the design to be modified relatively early into the design cycle moving the crossover point to higher and higher volumes, and also the shorter design period.

FPGAs are a trade-off solution to overcome the above problems, appearing the so-called "Structured ASIC approaches", which effectively take an FPGA design and replace it with the same logic but hardwired with a fixed routing. It is based on predefined sea of gates, with the interconnections added later, making them mask programmed FPGAs. Regarding to the fabrication process, it consists of adding from two to six metal layers forming the interconnection wiring between the gates. Due to the design and mask costs for the underlying silicon can be shared over a large number of devices, and only the metal interconnection layers need to be added to make a working device, the initial cost is much lower. Regarding the main vendors, Xilinx [194] and Altera [195] provide services based on their high-end FPGA families for converting an FPGA implementation into a structured ASIC. These devices outperform FPGAs due to important cost reductions for volume production, reaching a 50% improvement in throughput, with a cost less than half the power of the FPGA design they are based on [196]. Additionally, NEC Electronics Corporation also provides embedded gate array service [197].

In this subsection, some related motion estimation algorithms using specific circuits are shown in Table 2.4.

| Author | Family | Algorithm | fps (resolution) | Year |
|---|---|---|---|---|
| Warrington [198] | Matching | VBMSE | 30 (720×480) | 2007 |
| Verma [199] | Matching | Many | 30 – 60 (many) | 2008 |
| Sebastião [201] | Matching | Many | 25 (352×288) | 2008 |
| Ndili [202] | Matching | HMDS [203] | (352×288) (230kblocks/sec) | 2011 |
| Dhahri [204] | Matching | 4SST | (230kblocks/sec) | 2011 |
| Dhoot [178] | Matching | MC-TSST[178] | NP | 2011 |
| Stocker [205] | Gradient | Horn & Schunck | 5 (29×20) | 2006 |

*Table 2.4: State of the art motion estimation algorithms under ASICs.*

Warrington et al. [198] propose an ASIC architecture for high performance VBSME (Variable Block Size Motion Estimation) of the block-matching algorithm family, which supports MRF (Multiple References to Frames). For enabling better performance in terms of rate-distortion for different video contents, the architecture allows the selection of a spatial motion search in high resolution on a framework, or a search with MRF and lower spatial resolution. Comparing this system with an exhaustive search algorithm, it is obtained a reduction in the bitrate and the computational complexity. In addition, the implemented algorithm achieves similar state to the obtained with the exhaustive search, getting a performance of 30 fps with a resolution of 720×480.

Another interesting study is carried out by Verma and Akoglu [199], where it is presented a hybrid reconfigurable architecture on a coarse NoC38 mechanism. In this work, the implemented algorithms using VBSME have been FST, HS (Hexagon Search), HBS (Hexagon Big Search), SS (Spiral Search) [200] and DS. The proposed study design uses a high level of parallelism and intensive reuse of data. It also allows any size of block, which a priori is a problem from the perspective of an ASIC. Another interesting point is that the developed system requires a clock frequency above 0.8 MHz

to maintain 30 fps for intermediate frame sizes (176×144), and 29.16 MHz to support 60 fps for high definition (1280×720).

Meanwhile, Sebastião et al. [201] compared two implementations for encoding video, a FPGA and a ASIC, made both with IP cores[16]. It focuses on real-time motion estimation belonging to different families of block-matching algorithms (FST, TSST (Three Step Search Technique) and DS), since they represent the largest computational cost of such systems. In this case, the ASIC implementation is more suitable for devices that use batteries, while the reconfiguration ability of the FPGA allows the movement estimation which dynamically adapt the video encoder to the characteristics of the application. Their testing results have been reached using two different frame formats[17], CIF (352×288) and QCIF (176×144). Following, it is shown the maximum ratio achieved by frame in each case:

- For the FST algorithm, 2.07 fps in FPGA and 3.33 fps under ASIC are achieved for the CIF format, while 8.29 fps under FPGA and 13.30 fps under ASIC are achieved for the QCIF format.

- In the case of TSST algorithm, 21.12 fps (CIF) and 84.49 fps (QCIF) are achieved under FPGA, while under ASIC the ratios are 33.88 fps (CIF) and 135.52 fps (QCIF).

- For the ASIC implementation, the DS algorithm has ratios of 23.53 fps and 94.12 fps, for CIF and QCIF formats respectively, while 14.67 fps and 56.68 fps are the ratios for the FPGA implementation.

Ogunfunmi & Ndili [202] propose an efficient motion estimation algorithm [203] on a SoC[18] architecture in order to process high quality video in low power mobile devices. The achieved results are 15456.05 Kb/s in the case of an input sequence of 90 frames with size of 1280×720 (720p) each one.

[16] IP core definition: (http://en.wikipedia.org/wiki/semiconductor_intellectual_property_core)

[17] CIF and QCIF definitions: http://www.springerreference.com/docs/html/chapterdbid/10489.html.

[18] SOC definition: http://www.springerreference.com/docs/html/chapterbid/311302.html

Another example, the work of Dhahri et al. [204], proposes a parallel architecture for the 4SST (Four Steps Search Technique) motion estimation algorithm. This work develops a method that uses 9 elements processing, 2 local memories (one for the reference block and one for the search area), a unit of comparison, and counters to control memory addresses. Thus, it is allowed parallel processing pixels, and it is obtained results in real-time, without providing accurate obtained performance. Regarding the implementation of gradient models under ASIC architectures, Alan A. Stocker introduced into [205] a new VLSI (Very Large Scale Integration)[19] analog sensor that estimates the optical flow in two visual dimensions. Its computational network architecture consists of two layers of movement units, locally attached to jointly estimate the optimal optical flow field. For this purpose, the implemented algorithm is the Horn & Schunck [64] algorithm with a partial constraint which measures the distance of the optimal flow estimated with respect to a motion vector.

As the last example of block-matching algorithm applications, Dhoot et al. [178] explore low power motion estimation, specifically low power hierarchical search algorithms for motion estimation in the context of probabilistic computing. With the designed fault tolerant algorithm (MC-TSST) proposed in this paper, they show an increase of energy savings, that can be realized with probabilistic computing, of 70% versus to 57% with the conventional algorithm (TSST), achieving also minor impact on the quality of motion estimation. This work is focused on remarking the resilience of this system.

[19] VLSI definition: http://www.springerreference.com/docs/html/chapterdbid/311381.html.

# Chapter III

## Video compression and block-matching

This chapter presents a deep introduction to video compression and its standards, as well as a detailed presentation of block-matching algorithms emphasizing the used algorithms in this work.

It is also presented the error metrics used in this thesis and the input video sequences for the executed algorithms.

## 3.1. Video compression and its standards.

Video compression is developed around the issue of removing redundancy from the images that are going to be compressed. This data reduction is achieved thanks to the reduction of the number of bits used for codifying the video. Because every transmitted bit is expensive, compression is needed when transmitting information. In a vast majority of computer systems it is used the same amount of bits for coding one character or another, but regarding video compression, it is a tested fact that it is better to use less bits for coding the more frequent symbols.

There are a lot of encoding standards based on redundancy removal, of course, trying not to lose the information quality level. Two of the first advances in coding were the Shannon-Fano coding dating from 1940s [228 – 229] and the Huffman encoding scheme dating from 1952 [230]. These approaches have settled the bases for many video compression techniques which have played a key role in current multimedia systems.

In the current world, where the overall number of multimedia devices is growing very quickly in the portable device field, bandwidth plays a key role and so do video compression techniques too. Indeed, some compression techniques use the interlacing technique which reduces the required bandwidth without reducing the refresh rate. This is achieved scanning the video scene half of the time but displaying the video sequence at twice the scanning rate.

There are three main compression algorithm families, each one published by one of these three standardization organizations: the ISO (International Organization for Standardization), the ITU (International Telecommunication Union), and the MPEG (Motion Picture Expert Group) established in 1988 within the International Organization for Standardization (ISO) Steering Group (SG). As shown below, we can see the evolution of video compression standards from 1950 to beyond 2000.

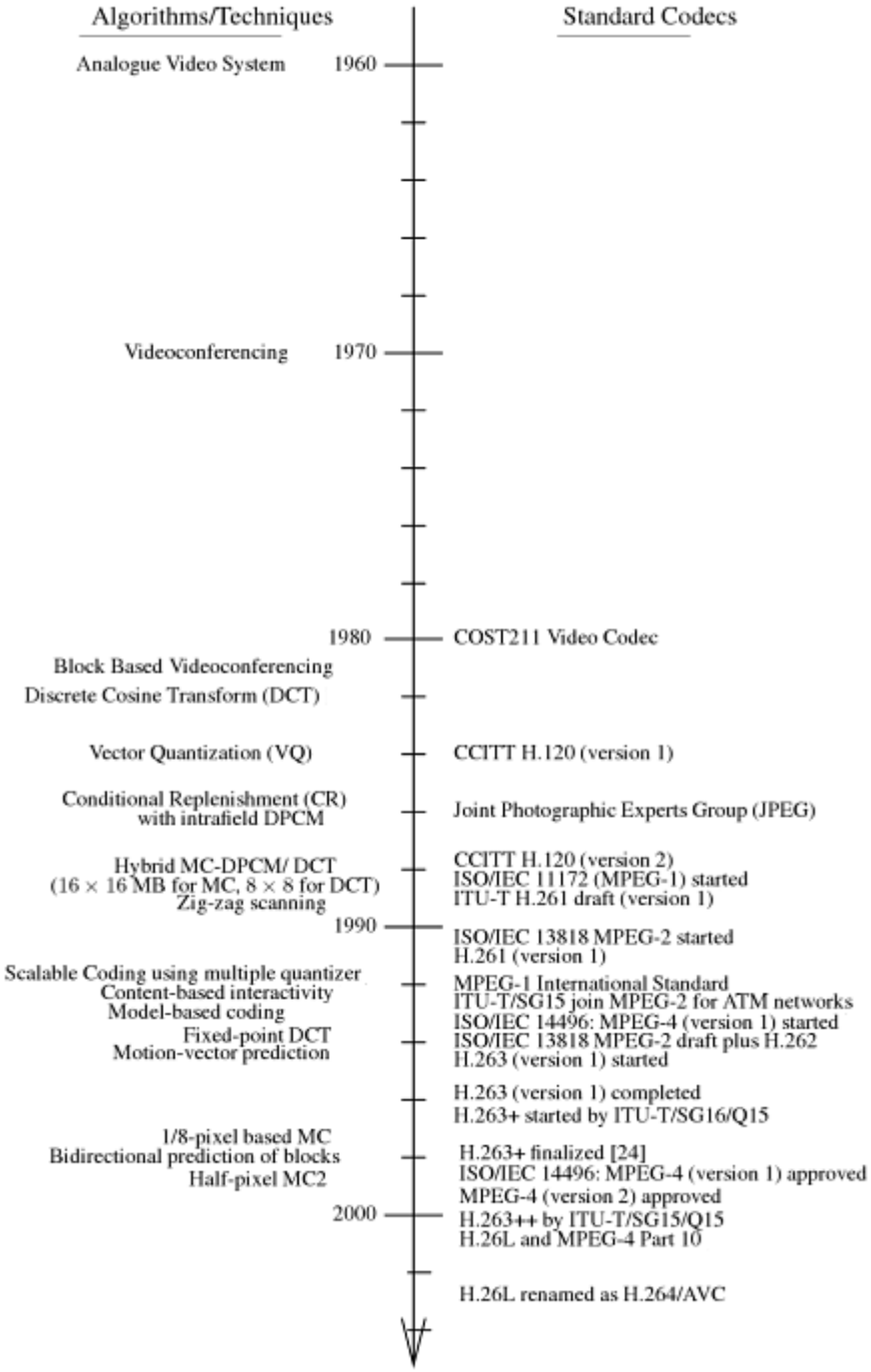


*Figure 3.1: Video standards evolution [206].*

Focusing on the image, we can assume the video compression evolution began at 1950s although the first analog video system was not tested until the 1960s. Around 1970, videoconference made its appearance in the multimedia world. It was in the 1980s when the video coding standardization models by the ITU started, by the name of CCITT (International Telegraph and Telephone Consultative Committee) [231]. In the

same decade, it appears the JPEG (Joint Photographic Experts Group) well known for the compressing static images algorithm. After this, around 1990 it appears the first MPEG video coding standard (MPEG-1 International standard) [232] which would be the first stone of one of the most popular video compression standard families. The video compression standard evolution reaches today bringing new standards like the ITU H.265 [233].

In Figure 3.2, we show the video standards and video formats which are mainly used nowadays in the framework of video coding.

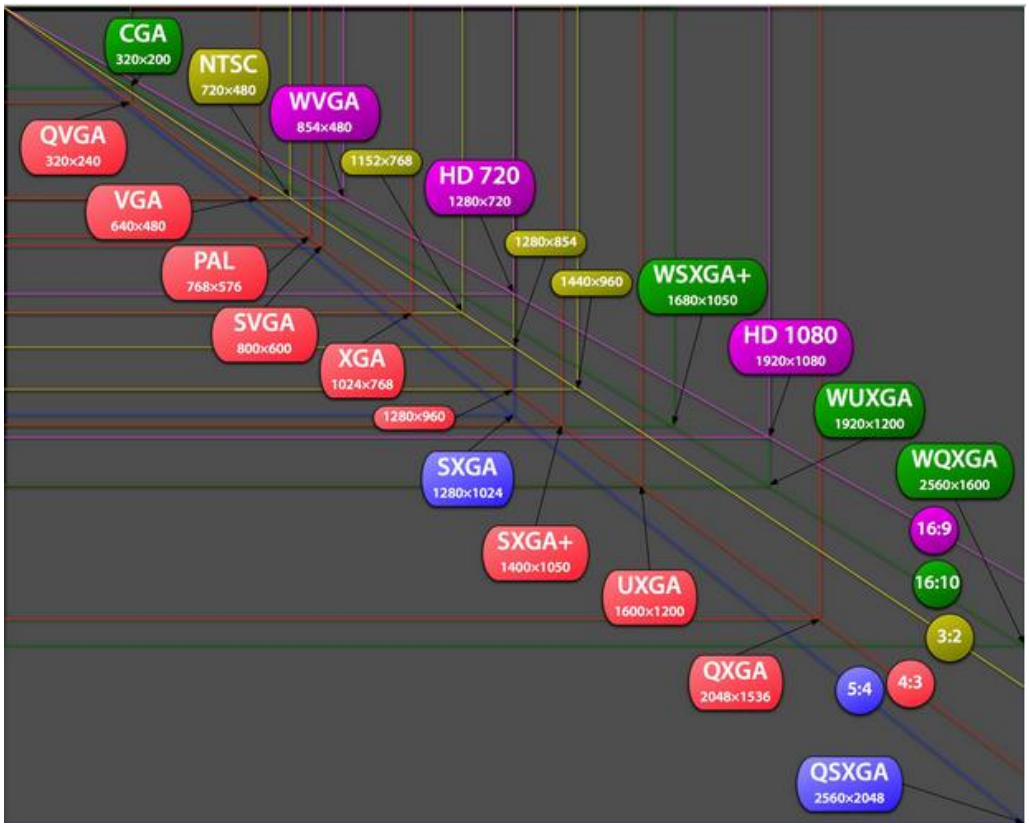


*Figure 3.2: Video standard and formats [207].*

### *3.1.1. Most used video standards.*

Among all standards shown above, we are going to focus on the currently more extended standards without focusing in the new H.265, which will be the center of the next point. These standards are the MPEG-4 [234] and the H.264 [235], both of which are going to be presented and described below.

On the one hand we have MPEG-4, the last video compression standard developed by the MPEG (Motion Picture Expert Group) as an evolution of its previous older brothers MPEG-3 [236], MPEG-2 [237] and MPEG-1 [232]. Indeed, it supports all the functionalities included in them, and covers digital audio and, video presentation and transmission. The MPEG-1 aims to meet the low complexity requirement; MPEG-2 is meant for broadcast-quality television, and MPEG-3 as an improved version of MPEG-2, is used to encoding HDTV (High Definition Television) [238]. All these standards form a standard family with a very wide bitrate range which varies from 4800 bit/s to 4 Mbit/s. Its last member, MPEG-4, entails some new features like supporting for 3D rendering, digital rights management, or other interactive applications. Main applications which use MPEG-4 are found in videophone, broadcast television, or multimedia streaming.

MPEG-4 is made around a "core" video encoder/decoder based on the DPCM/DCT (Differential Pulse Code Modulation/Discrete Cosine Transform) [239 – 240] coding algorithm. This video encoder/decoder is helped by a set of video compression tools based on the ITU-T (International Telecommunications Union - Telecommunication Standardization Sector) H.263 standard, which is more efficient than MPEG-1 and MPEG-2. We have to highlight some MPEG-4 features:

- High compression efficiency for storing and transmitting high-quality video sequences.

- Content-based interactivity allowed by a new concept in MPEG-4, which is basing video in objects rather than frames, VOP (Video Object Plane) [241].

- Universal access for multiplatform and multi-device use.

Focusing on the coding feature we have to detail that MPEG-4 introduces another new concept, the use of irregular-shaped objects, allowing the distinction between foreground and background. In contrast, MPEG-4 video codec only supports YUV [4:2:0] 176×144, QCIF (Quarter Common Interface Format), or 352×288 CIF (Common Intermediate Format) video representation regarding compression context.

In Figure 3.3 we show an example of the context-based approach used by the MPEG-4 standard.

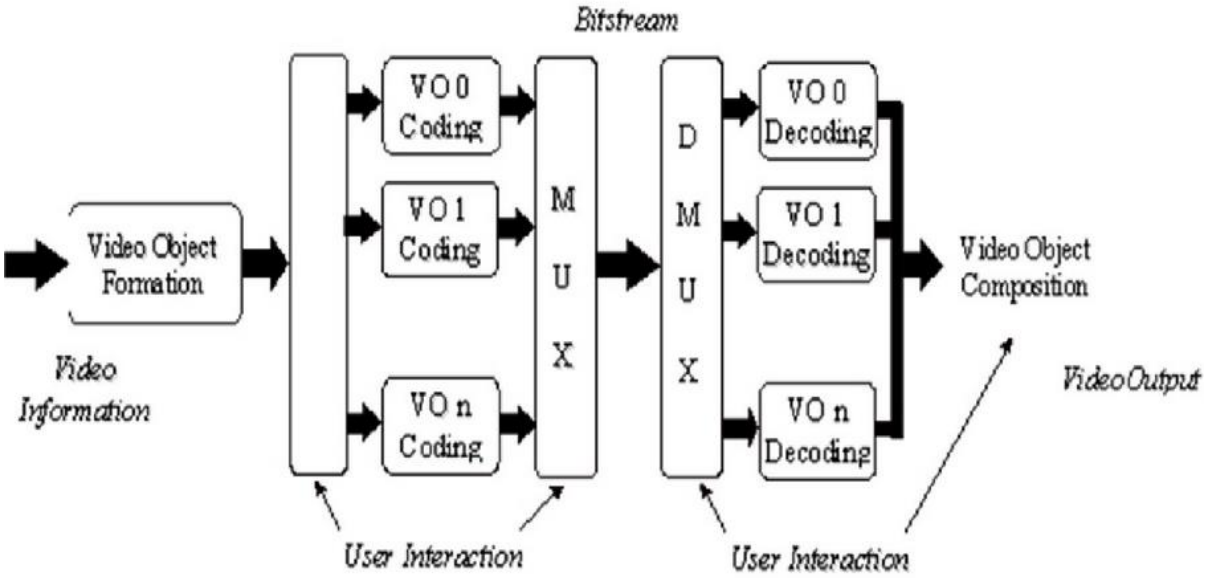


*Figure 3.3: MPEG-4 coding/decoding flow [208].*

On the other hand we have the H.264 (also known as MPEG-4 part 10 and formerly known as H.26L [242]) standard which belongs to a cooperation between the ITU (International Telecommunication Union) VCEG (Video Coding Experts Group) and the ISO (International Organization for Standardization)/IEC (International Electronic Commission) MPEG committee. The H.264 standard improves even more the coding efficiency with the same video quality than MPEG-4 or H.263. Moreover, it is improved for packing the coded data in a better way for transmitting over the network. Because of this, it is widely used in real-time videoconference, storing, broadcasting, or streaming video applications. Although the H.264 encoder inherits some coding stages like prediction, transform, quantization, and entropy from other previous standards like MPEG-1 [232], MPEG-2 [237], MPEG-4, H.261 [243], or H.263 [244], it has several improvements, like for example, the use of two different datapaths (forward and reconstruction), spatial prediction in intra-frame coding, or motion compensation with adaptive block size. H.264 is one of the most accurate current coding standards; it uses quarter-pixel accurate motion compensation, although it could be better achieving eighth-pixel accuracy or even more. Also, the H.264 has the option of having several reference frames in its inter-frame coding, which makes it more error resistant.

H.264 employs a deblocking or smoothing filter to avoid the consequences of the 2D video frame truncation before the DCT (Discrete Cosine Transform). Another peculiarity of H.264 is the use of three different transforms depending on the type of MCER (Motion Compensated Error Residual). Almost every previous standards use the 8×8 pixel DCT, like MPEG-1, MPEG-2, MPEG-4, and H.263 [244]. But the H.264 uses transformation of 4×4 pixel luminance discrete cosine coefficients in intra-frame macroblocks, transformation of 2×2 array of chroma discrete cosine coefficients, or transformation of all other 4×4 blocks depending on each case MCER. For the quantization process it uses a scalar DCT coefficient compressing its results with the well-known VLC (Variable Length Coding) [245].

Because of these main features and others, H.264 is optimized for compression, which makes it very powerful for multimedia communications.

To see compared both presented standard we show in Figure 3.4 and Figure 3.5 the both encoding process block diagrams.

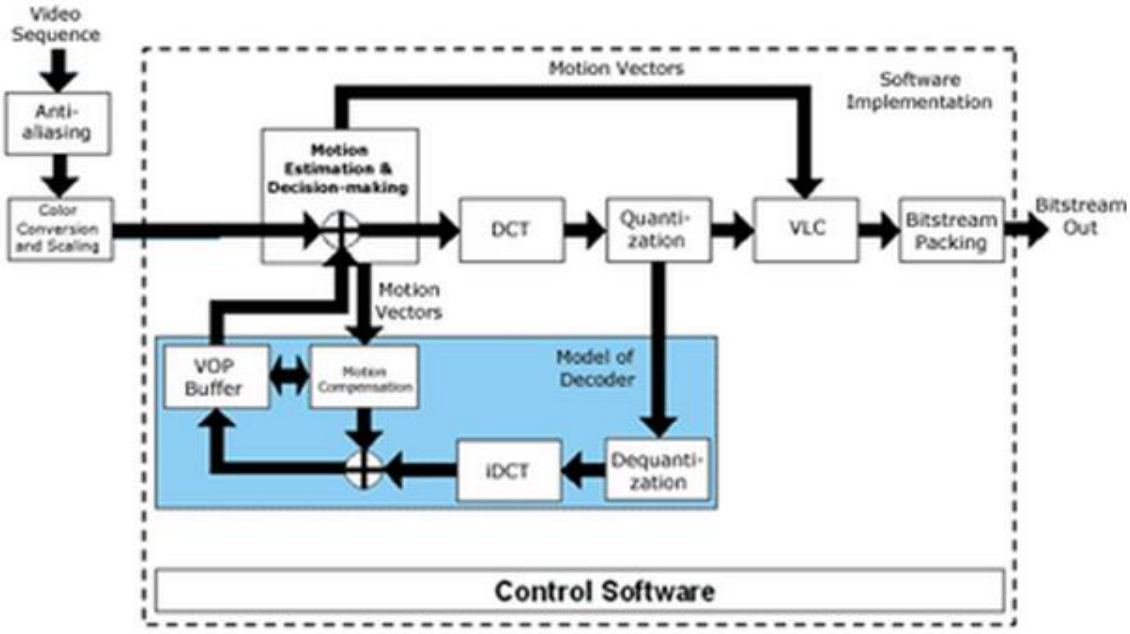


*Figure 3.4: MPEG-4 encoding process [209].*

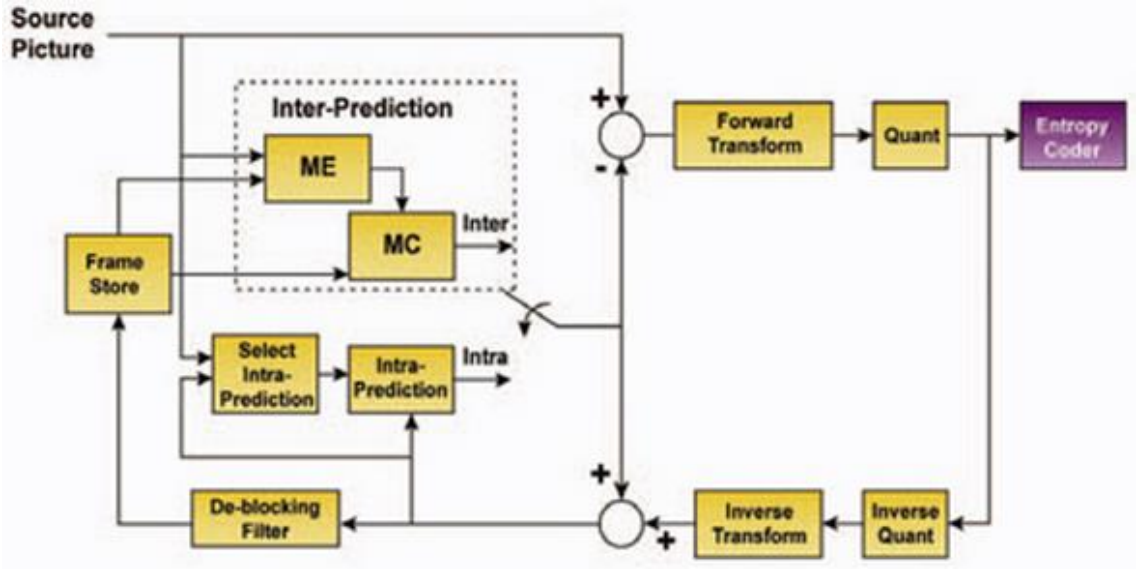


*Figure 3.5: H.264 encoding process [210].*

### *3.1.2. H.265 (new standard).*

HEVC (High Efficiency Video Coding), also known as H.265 [233], is the next generation video compression standard after H.264, finalized in April 2013 and developed by the ITU VCEG and the ISO/IEC MPEG. Due to the increasing popularity of HD Video (High Definition Video), and the request of higher resolutions overall in the Internet networks, the H.264 standard has had to be advanced by its bigger brother the H.265 standard.

While the H.264/MPEG-4 standard was spreading out in the multimedia field, the VCEG and the MPEG groups continued looking for a fastest standard. This work from these two groups would lead them to create in January 2010 the JCT-VC (Joint Collaborative Team on Video Coding). And as its first goal, the starting of the H.265 project. Continuing with the meeting of this new group, this standard was being shaped and developed for giving a stable standard at the end of 2012. This standard was issued in January 2013 and is formally known as ITU-T H.265 or ISO/IEC 23008-2.

At a high level, H.265 is very similar to its predecessor H.264, but they are very different in almost all the elementary steps [213]. Indeed, H.265 is based on the well-known block-based hybrid video coding scheme including closed loop motion compensation and transform coding. It is also based in variable block size prediction

and compensation achieving high image quality with a remarkable bit reduction. Moreover, H.265 uses three frame types, I (Intra)-, B (Bidirectional)- and P (Predictive)-frames within a group of pictures, using inter-frame and intraframe compression, but also, it carries some new features seen below [211].

Before explaining these features we have to define some concepts about H.265: the CU (Coding Unit), the PU (Prediction Unit), and the TU (Transform Unit) [213]. The CU is the basic coding unit similar to the H.264/AVC's macroblock, it has to be square but it can be of different sizes. The PU is the basic unit for prediction and the TU is the basic unit for transforming and quantizationing. The CU allows in intra-prediction mode recursive splitting into four equally sized blocks. It starts with the LCU (Largest Coding Unit), 64×64, to finalize with the SCU (Smallest Coding Unit), 8×8. The PU is the same size as CU and can be further split when CU size is 8×8. It is possible to merge adjacent PUs not necessarily of rectangular shape as said of CUs. The TU is similar in concept to the DCT transform, but really is an integer spatial transform which range varies from 4×4 to 64×64. Moreover, for block sizes bigger than 8×8, a rotational transform can be applied to lower frequency components. And now, these are the H.265's new features:

- CTB (Coding Tree Blocks): instead of using 16×16 pixels macroblocks as H.264 standard, H.265 uses 64×64 pixels CTBs. This is due to the fact that large frame sizes are more efficiently encoded using larger block sizes, for example in 4K resolution. In Figure 3.6, we can see the difference of using this new technique in H.265 against H.264.

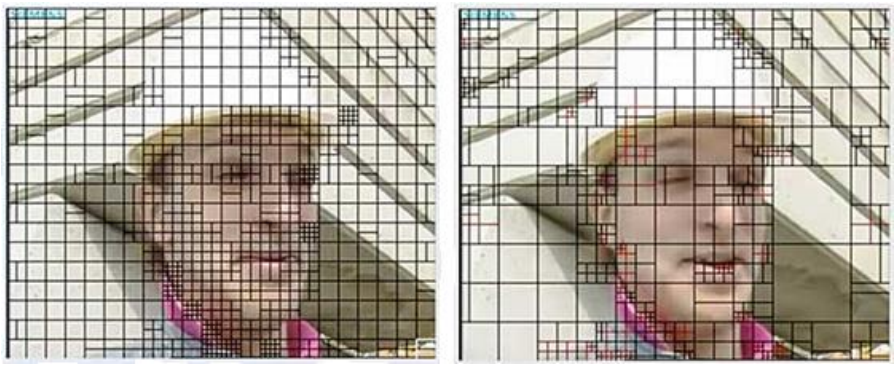

*Figure 3.6: H.264 (left) vs H.265 (right) [211].*

- More intra-prediction directions: H.265 intra-prediction technique joins two simplified directional intra-predictions methods, the ADI (Arbitrary Direction Intra) and the AIP (Angular Intra Prediction), using around 35 prediction directions instead of the nine directions that H.264 uses. This leads to a lower-complexity method in which parallel processing can be achieved allowing a higher intra-frame compression. This new feature can be seen in Figure 3.7.

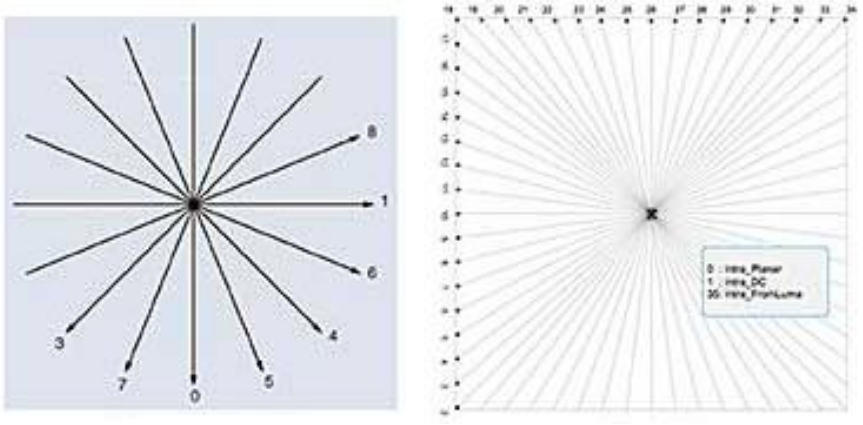


*Figure 3.7: H.264 (left) vs H.265 (right) [211].*

- Improved deblocking filter: within the prediction loop it operates a deblocking filter similar to the used at H.264, but improved for this new standard.

- Parallelization tools available for multi-core environment.

- Entropy Coding in H.265: H.265 defines two entropy coding patterns depending on the complexity, one for higher and one for lower. The one used for the lower one is very similar to the CALVC pattern of H.264, but in this case, a re-sorting of code table elements can be used as a compression improvement aid. The one used for the higher one is very similar to the H.264 CABAC coder but using variable length instead, allowing a higher throughput per processing cycle than seen CABAC [212][233].

Now, regarding the results of the H.265 standard we can see the improvement done by the JCT-VC group saving a 35.4% in the bitrate compared to its immediate predecessor the H.264 [211 – 212]. Also, we can see the increasing measurements that Table 3.1 arises about the improvements achieved by the H.265 standard against its not immediate predecessors.

AVERAGE BIT-RATE SAVINGS FOR EQUAL PSNR FOR ENTERTAINMENT APPLICATIONS

| Encoding | Bit-Rate Savings Relative to | | | |
|---|---|---|---|---|
| | H.264/MPEG-4 AVC HP | MPEG-4 ASP | H.263 HLP | MPEG-2/ H.262 MP |
| HEVC MP | 35.4% | 63.7% | 65.1% | 70.8% |
| H.264/MPEG-4 AVC HP | – | 44.5% | 46.6% | 55.4% |
| MPEG-4 ASP | – | – | 3.9% | 19.7% |
| H.263 HLP | – | – | – | 16.2% |

*Table 3.1: H.265 improvements against its predecessors [211].*

## 3.2. Block-matching algorithms.

In this section we are going to provide an overview of the matching algorithms focusing on its two bigger families. Like we already know, the aim of block-matching algorithms is to estimate motion vectors for our current frame comparing the current macroblock to each macroblock within a specific and fixed search window in the reference frame [246 – 247]. So, if we do this in the most simple approach, we would do an exhaustive search algorithm, matching all macroblocks within a search window in the reference frame to estimate the optimal macroblock; i.e., the one with the minimum BME (Block-Matching Error). There are several definitions for BME, but the most used are the SAD (Sum of Absolute Difference), for every pixel between a MB (MacroBlock) of the current frame and a MB of the reference frame, and the MSE (Mean Squared Error), being this less conservative due to the square factor. Then, the huge amount of computations required calculating the error by these algorithms is too high, so in order to optimize motion estimation calculation, many enhanced search algorithms have been proposed. These methods can be organized in two main families:

- SR (Search reduction): these techniques are based on reducing the search points within a search window following the pointed algorithm, which in most cases moves or changes the search window for improving its accuracy [248 – 251].

- CR (Calculation Reduction): Conversely, algorithms categorized as CR of SAD try to reduce the computations. Since SAD is calculated by adding the differences of each pixel, the computation of the partial SAD is simpler than the computation of the

total SAD between two MBs [252]. The idea thus, is early reject the invalid motion vectors by means of not comparing all macroblocks available.

### *3.2.1. SR (Search Reduction).*

Following, the four main techniques inside the search reduction family are going to be detailed, and later, other important algorithms inside this family will be presented.

- ***FST (Full Search Technique).***

This algorithm is the most intuitive, the most straightforward, and the most accurate one, because it matches all the possible blocks within the search window in the reference frame for finding the one with the minimum SAD (Summation of Absolute Differences), defined in expression 3.1:

$$SAD(x,y;u,v) = \frac{1}{N^2}\sum_{x=0}^{N-1}\sum_{y=0}^{N-1}\left|I_t(x,y) - I_{t-1}(x+u,y+v)\right| \tag{3.1}$$

Where $I_t(x,y)$ is the pixel value in the coordinate $(x,y)$ in the frame $t$, and $(u,v)$ represents the motion of the candidate macroblock.

Giving an example, with a block with size N = 32, the FST requires 1,024 subtractions and 10,023 additions to calculate the SAD. Moreover, $(1+2d)^2$ blocks are checked while the search window is limited within $\mp$ d pixels, usually by a power of two.

In Figure 3.8, we can see on the left one block which is part of frame T. This block is being matched with the corresponding one on the right part of frame T + 1 inside the search window using an error metric like SAD for calculating the similitude. So, the displacement from the block in frame T to the block in frame T + 1 constitutes the estimated motion for this block.

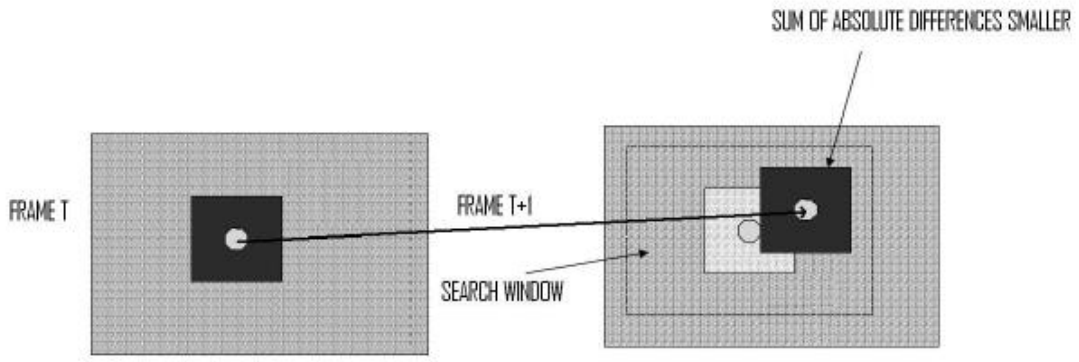


*Figure 3.8: FST process [214].*

- ***2DLOG (Two Dimensional Logarithmic Search).***

The 2DLOG (Two Dimensional Logarithmic Search) [253] appears like an alternative to the NSST (N Step Search Technique) by Jain and Jain in 1981. It starts with a rough search grid, and it continues refining it when the current estimated vector is the center of the current search area.

The 2DLOG technique uses a pattern cross (+) in each step with an initial step size of d/4. The step size is divided by half only when the point with the minimum SAD of the previous step is the center or reaches the search window boundary. In other case, the step size remains fixed. When the step size is reduced to one, all eight of the checking points adjacent to the center checking point of this step are searched.

In Figure 3.9, we can see an example of using the 2DLOG search technique. The first step (blue color) achieves the best point at the top. Following the step two (green color) achieves the best point on the right as the third step (yellow color) does. Fourth step (red color) achieves the best point on the center, so step size is reduced to half. Next, the fifth step (brown color) reduces the step size by half because the boundary is reached. Finally (grey color), a one size step is taken.

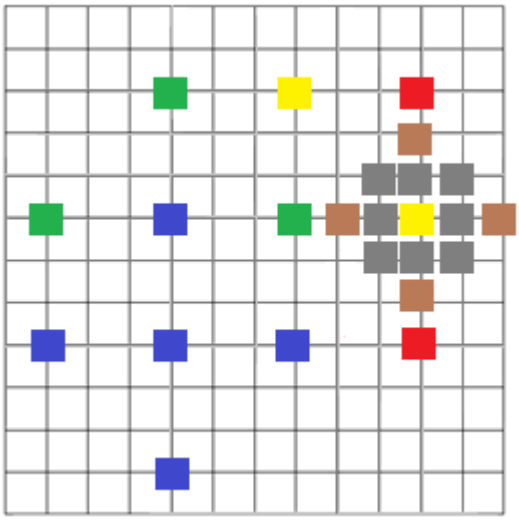

*Figure 3.9: 2DLOG process.*

One of the main drawbacks of this algorithm is that its complexity cannot be perfectly managed because of the number of iterations is variable.

- ***NSST (N Step Search Technique).***

The NSST has a lot of variations but all of them derive from the original idea, the Three Steps Search Technique (TSST) [254]. The TSST supports two important contributions for motion estimation in terms of fixed search patterns and limited search steps. Because the TSST was the first block-matching technique with a non-exhaustive search, many later works have included these characteristics for their design.

The NSST key idea is to perform a multi-scale process, applying "N" steps in order to find the most similar macroblock within the search window of the reference frame. We are going to focus and describe the TSST which is the basic case of the NSST. In the TSST first step, the step size of the search window is designated as half of the search area. Then, nine candidate points which are the center and eight checking points on the boundary of the search window, as shown in Figure 3.10 (blue color), are selected in each step. The second step moves the search center towards the matching point with the minimum SAD of previous step, reducing later the step size to the half as shown in Figure 3.10 (green color). In the third step, the search process is stopped, remaining to one pixel the step size and obtaining the optimal motion vector with the minimum SAD, as shown in Figure 3.10 (yellow color).

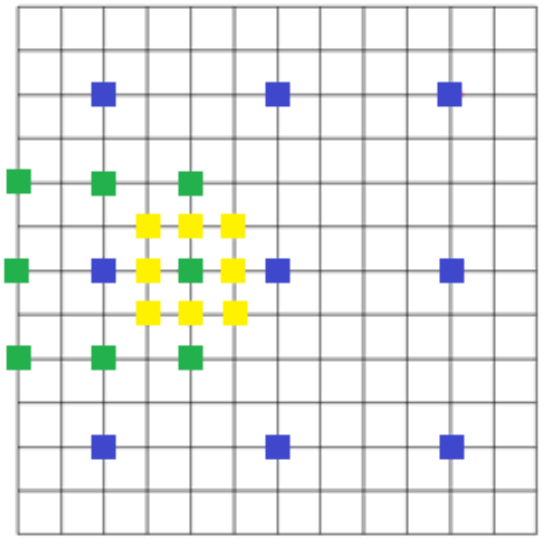

*Figure 3.10: TSST process.*

So, once described the basic case of the NSST we describe the NSST like a N – 2 iterations of the first step followed by the second and third steps of the TSST.

Comparing FST versus TSST using the same search window, $\mp$ 7 pixel, the TSST only needs 25 search points in comparison with the FST algorithm, which needs 255.

In Table 3.2, we show a comparison between FST, TSST, and 2DLOG algorithms, showing that FST uses more search points than TSST, and this one, more than 2DLOG.

| | Number of Search Points | | Number of Search Steps | |
|---|---|---|---|---|
| **Method** | **MIN** | **MAX** | **MIN** | **MAX** |
| FULL SEARCH (EXHAUSTIVE) | 225 | 225 | 1 | 1 |
| THREE-STEP SEARCH | 25 | 25 | 3 | 3 |
| 2D-LOG SEARCH | 13 | 26 | 2 | 8 |

*Table 3.2: Candidate points for FST, TSST, and 2DLOG [214].*

There are some variations of the basic TSST. We are going to talk briefly about two of them.

On the one hand, we have the NTSST (New Three Steps Search Technique) [255 – 256], which exploits the fact that the motion vectors of the frame with less motion are mostly found near the center of the search window, allowing an early termination of this technique. For improving the TSST performance, the NTSST manages a center biased

checking point pattern and a halfway-stop for stationary macroblocks. In the first step, the NTSST behaves like the TSST, checking the nine candidate points. In the second step, the NTSST ends whether the center point contains the minimum SAD; otherwise, it checks five corners or three edge points. The rest of steps of the NTSST are similar to those of the TSST.

On the other hand, we have the 4SST (Four Steps Search Technique) [257], which is based on the TSST but reduces the complexity of it by checking only a subset of the candidate points. In the first step (blue color), it fixes a step size of two and searches in nine candidate points (center and eight boundary points). If the best point (the one with the minimum distortion) is the center then it goes to the last step, otherwise, the best point is the new center. In the second step (green color), the number of candidate points depends on the pattern which depends on the previous best point, as shown in Figure 3.11. The third step (yellow color) behaves identically to step two but using another pattern, and finally the fourth step (red color), fixes the step size to one pixel and searches nine candidate points (center and eight boundary points) for finding the best one.

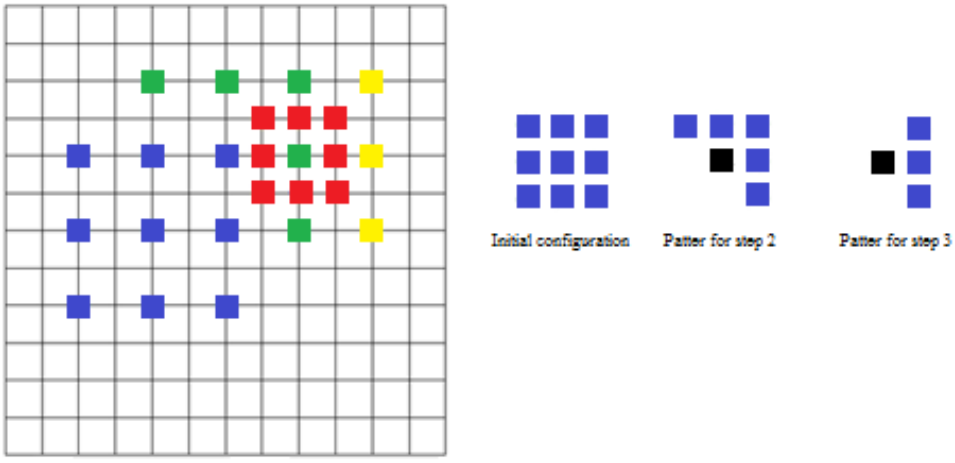


*Figure 3.11: 4SST process.*

- ***DS (Diamond Search).***

The DS (Diamond Search) [258] designed by Zhu, S, and Ma in 2000 is based in two search different patterns. Both are shown in Figure 3.12 where the left one is the LDSP (Large Diamond Shape Pattern) and the right one is the SDSP (Small Diamond Shape Pattern).

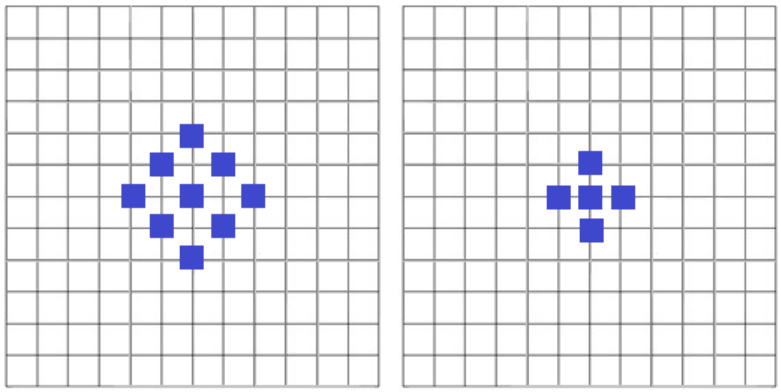

*Figure 3.12: LDSP (left) and SDSP (right).*

At its first step, the DS algorithm used the LDSP and depending on the best point, the LDSP or the SDSP is used in the second step. In the second step, it uses the SDSP if the best point is the center; otherwise, it uses a superposed LDSP over the previous one for not checking previously checked points. The algorithm stops when the best point of the SDSP has been found. In Figure 3.13 we can see a demonstration of the DS process with a first step (blue color), a second step (green color), and a third and final step (yellow color).

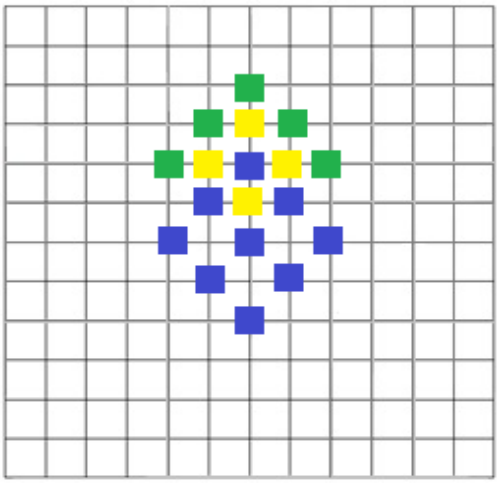

*Figure 3.13: DS process.*

DS has very good performance due to the fact that it has neither a too small search pattern nor a too large one. Another characteristic is that the search area is not limited. DS has a similar performance to NTSST but with a computation reduction greater than 20% [259]. Because of its effectiveness, it has been integrated in the reference software of the MPEG-4 video coding standard.

- ***Other algorithms.***

Although we have presented the main SR (Search Reduction) algorithms there are many other techniques, but we are going to mention only three of them.

- BS (Binary Search): it is used by the MPEG-Tool and based on the idea of dividing the search window into a number of regions and doing a FST only in one of these regions [260]. In its first step, it calculates the MAD (Mean Absolute Difference) on a grid of 9 pixels that include the center, the four corners of the search window, and four points at the boundaries. Based on these points, the search window is divided into regions. In the second step, it processes a FST in the region with the minimum MAD.

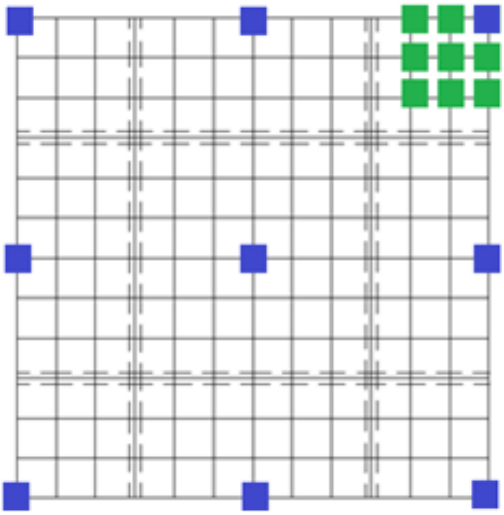

*Figure 3.14: BS process.*

- SS (Spiral Search): it was proposed by Zahariadis and Kalivas in 1995 combining TSST and BS [260]. In its first step, the step size is fixed to half the maximum displacement in the search window. Then, nine candidate points (five from a cross (+) around the center of the search window, and four from the corners of the search window) are chosen for calculating the best one. In the second step, the step size is reduced by the half and the best point is searched between nine candidate points (center and eight boundary points like TSST). This step is performed until step size achieves one.

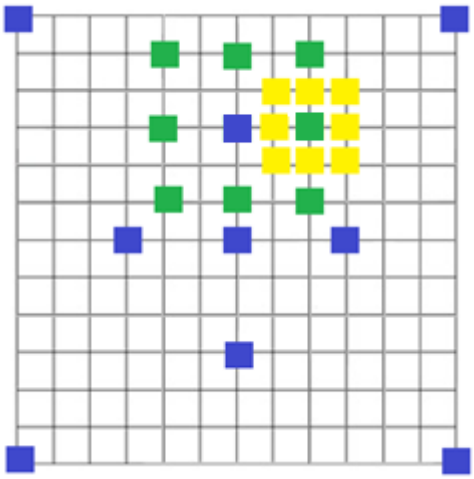

*Figure 3.15: SS process.*

- HS (Hexagon Search): it was proposed by Zhu, Lin and Chau in 2002 [261]. It is very similar to the DS algorithm but using different patterns. Its main advantages against DS are its diagonal speed, $\sqrt{5}$ instead of 2 in the DS case, and the number of checked points at each new iteration, which is only three points in the HS case. Due to these advantages its complexity is reduced nearly 40% against DS. As shown in Figure 3.16, HS is based on two different patterns, LHP (Large Hexagon Pattern) on the left, and SHP (Small Hexagon Pattern).

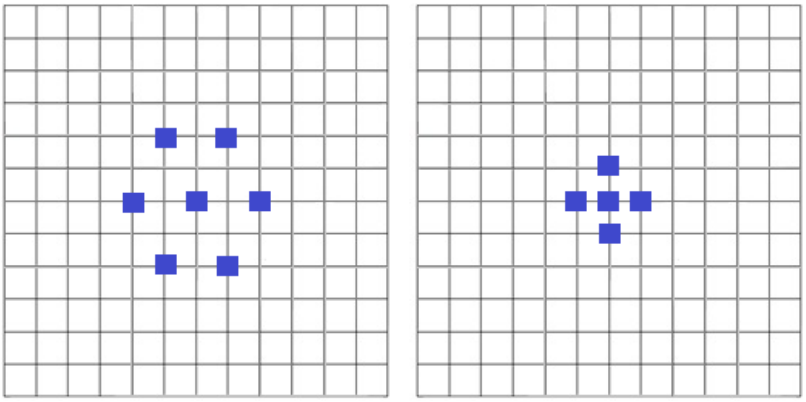

*Figure 3.16: LHP (left) and SHP (right).*

Due to its performance the HS algorithm is integrated into the H.264/MPEG-4 AVC standard.

In Figure 3.17, we can see a demonstration of the HS process with a first step (blue color), a second step (green color), and a third and final step (yellow color).

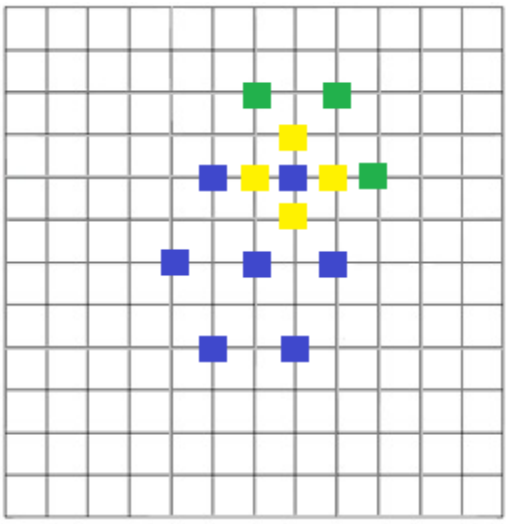

*Figure 3.17: HS process.*

### 3.2.2. CR (Calculation Reduction).

Because of the fast increase of streaming media, computational complexity is being an important point to improve when talking about motion estimation. This fact is being attack by CR motion estimation algorithms family. Opposite to SR (Search Reduction) algorithms, CR (Calculation Reduction), also known as CR (Computation Reduction), algorithms are based on the idea of reducing the number of computations instead of the SAD itself. Since SAD is calculated by adding the differences between a pair of pixels, the computation of the partial SAD is simpler than the computation of the complete SAD between two macroblocks. This is the key idea in the CR algorithms. Following, the two main techniques inside the CR algorithms family are going to be detailed.

- ***PDST (Partial Distortion Search Technique).***

The PDST [220 - 221] is based on the idea of removing unnecessary computations in an effective way, being easily realized in VLSI (Very Large Scale Integration) systems. The way of removing the unnecessary computations is achieved by finishing the measuring distortion calculation, every period complete SAD, when the partial SAD is already greater than the minimum SAD found at the current search process.

Regarding block-matching, we have a block of M×M pixels. Then, we split this macroblock into “k” different small pixels groups so that the distortion of the “kth”

group is "dk". If we were going to calculate the partial SAD during the matching process, we could describe the distortion of the "kth" group as shown in Formula 3.2:

$$d_k = \sum_{i=1}^{k} \sum_{j=1}^{M} \left| f_t(x+i, y+j) - f_{t-1}(x+i+mx, y+j+my) \right| \tag{3.2}$$

Where "$k$" is the number of pixels groups, $M$ is the size of the block, $(x, y)$ is the position in the macroblock, "$mx$" and "$my$" are the horizontal and vertical components of the candidate motion vector, $f_t(.,.)$ represents the luminance pixel intensity of the current frame and $f_{t-1}(.,.)$represents the luminance pixels intensity of the reference one.

Usually, every period matching error accumulated as SAD is completely calculated for being compared later to the minimum SAD achieved; but in PDST the partial SAD is calculated and compared in every step to the minimum SAD already found, which corresponds to another candidate motion vector. If the partial SAD calculated in any step is higher than the minimum SAD already found, the candidate macroblock would not be the best candidate regardless the rest of the incomplete matching computations. Therefore, PDST is able to erase impossible candidates before calculating their whole SAD reducing considerably the number of made calculations. This will reduce not only the number of subtractions and additions, but also, the number of times reading memory.

- ***NPDS (Normalized Partial Distortion Search).***

The Normalized Partial Distortion Search (NPDS) [222] technique is another algorithm from the CR family. This technique is based on the previous defined PDST, using also a halfway stop technique in the distortion calculation, but with an important difference. In NPDS the accumulated partial distortion, and also the current minimum distortion are normalized, so the impossible motion vector candidates rejected are increased.

Due to the increased number of comparison operations done in PDST, in NPDS the partial distortion is made over group of pixels instead of over single pixels. So, the macroblock is divided into pixels groups where each group consists of 16 adjacent pixels. The calculation order of the distortion between pixels follows a specific order

designed for ensuring that the pixels considered for the distortion calculation are equally distributed into the block. In Figure 3.18, we can see both, the macroblock division, and the pixels order followed to calculate the distortion, taking the upper left corner group of pixels as example.

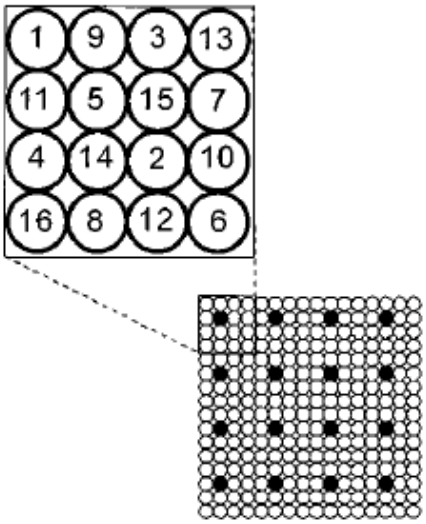


*Figure 3.18: Followed order (left) and macroblock division (right) [222].*

The NPDS algorithm behaves like the FST matching all the checking points inside the search window. The search begins at the origin point, moving later outwards following a spiral scanning path; in this way, it exploits the center-biased motion vector distribution characteristics of the real world video sequences, as shown in Figure 3.19.

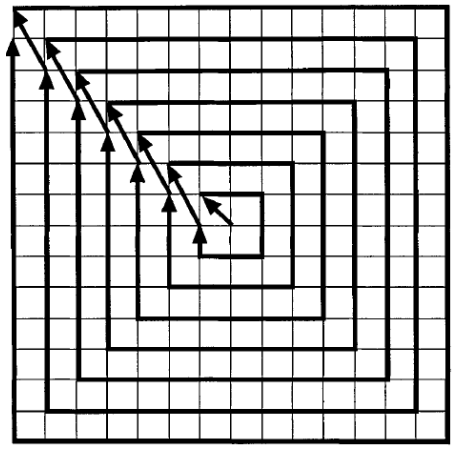

*Figure 3.19: NPDS search pattern [222].*

Behaving like PDST, the NPDS algorithm compares each accumulated partial distortion with the minimum achieved distortion; but in its case, with the normalized minimum distortion instead of the minimum distortion. This comparison is made by an integer comparison between the normalized versions of the accumulated partial distortion and the minimum partial distortion achieved at that moment.

- ***Other algorithms.***

We have presented the main CR algorithms. There are many other techniques, but we are going to mention only three of them.

- APDS (Adaptive Partial Distortion Search): this algorithm tries to improve one of the PDST defects, which is the speed for detecting the impossible candidates. Pixels with high activities, like edges and textures, contribute in most manners to the SAD criteria, because these kinds of pixels are very representative in the macroblock. So based on that, APDS [223] focus on computing the accumulated partial distortion of the representative pixels before other pixels. With this, the APDS algorithm reduces the algorithm computational complexity without decreasing the quality of the predicted image. For getting the higher activity pixels accumulated distortion in advance, the ADPS follows the Hilbert scan of the macroblock; which scanning a 2-D image extract a 1-D sequence of pixels preserving the 2-D coherence. An example of the Hilbert scan for a 16×16 pixels macroblock is shown in Figure 3.20.

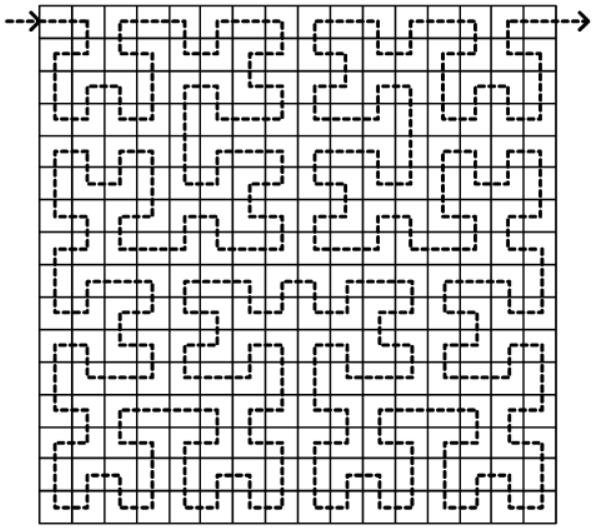

*Figure 3.20: Hilbert scan [223].*

- HPDS (Hierarchical Partial Distortion Search): it is based on a hierarchical approach so that a group of candidate motion vectors is selected to enter to the next level refining the search process [224]. This algorithm is divided into three levels. In the first level, every position in the search area is processed using a M:1 (M as size) decimation partial distortion function; although for reducing the computational cost, the area is divided into four groups, as shown in Figure 3.21, so that candidate vectors are selected for each group.

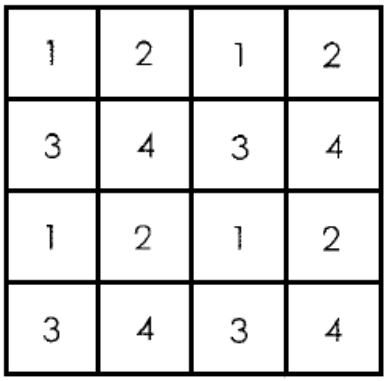


*Figure 3.21: Search area group division [224].*

In the second level, candidate motion vectors are searched using 2:1 decimation partial distortion function getting a group of motion vector with minimal partial distortion. In the third and final level, it is used all the block pixels for calculating the BDM (Block Distortion Measure) and choosing the output motion vector.

- PPDS (Progressive Partial Distortion Search): it tries also to improve the PDST computation cost increasing the early rejection rate of impossible candidates. Indeed, PPDS [225] technique increases in the best case the maximum computational reduction to 64 times against 16 times in the NPDS. The PPDS algorithm divides the processed macroblock into "P" equal size partial distortions and uses normalized accumulative partial distortion, as does NPDS algorithm, for rejecting the impossible candidates when the accumulative partial distortion is greater than the normalized minimum distortion achieved. In Figure 3.22 (from (a) to (d)), they are shown for the first partial distortion different pixels groups of 1 pixel, 2 pixels, 4 pixels, and 8 pixels respectively.

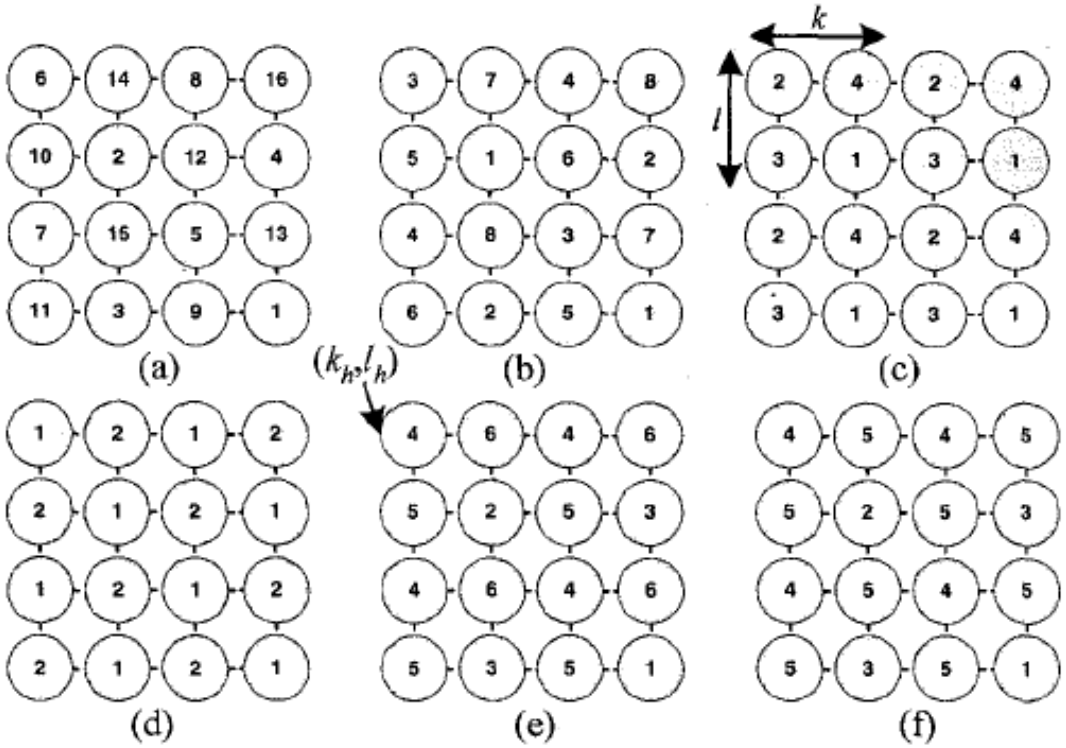


*Figure 3.22: PPDS pixel groups (a to d) and patterns (e and f) [225].*

Also, it is shown in Figure 3.22 (from (e) to (f)) progressive regular patterns used in this PPDS algorithm.

### *3.2.3. Time complexity.*

We compile the computational complexity of every presented algorithm in Section 3.2, both the search reduction block-matching algorithms and the calculation reduction block-matching algorithms.

In this way, we have integrated their computational complexities in Table 3.3 as resume.

| Algorithm | Maximum search points | Computational complexity |
|---|---|---|
| FST [214] | $(1+2d)^2$ | $O(N)^2$ |
| 2DLOG [215][253] | $2+7log_2 d$ | $O(log_2 N)$ |
| TSST [226][254] | $1+8log_2 d$ | $O(log_2 N)$ |
| DS [226][258] | $9+5log_2 d$ | $O(log_2 N)$ |
| BS [260] | $(d/2)^2$ | $O(N)^2$ |
| SS [260] | $9+8log_2 d$ | $O(log_2 N)$ |
| HS [226][261] | $7+3log_2 d+4$ | $O(log_2 N)$ |
| PDST [220 – 221] | $(1+2d)^2$ | $O(N)^2$ |
| NPDS [222] | $(1+2d)^2$ | $O(N)^2$ |
| APDS [223] | $(1+2d)^2$ | $O(N)^2$ |
| HPDS [224] | $(1+2d)^2$ | $O(N)^2$ |
| PPDS [225] | $(1+2d)^2$ | $O(N)^2$ |

*Table 3.3: Time complexity, being "d" the displacement.*

## 3.3. Benchmarks and error metrics.

This section presents the test images used in the different stages of this work. Not every presented image has been used in every stage, but every image sequence shown in the next point has been used for testing any improvement achieved in this work, applied to different block-matching algorithms. Also, these images have been used for presenting the results of the different experiments realized across the different stages of this thesis. Apart from the image sequences, this chapter presents the different metrics used in this work, being error metrics or throughput metrics.

### 3.3.1. Test images.

In this subsection, it is going to be presented and discussed the different sequences of images that have been used for testing, measuring, and comparing the different block-matching techniques improved in this work with the different proposed techniques. Moreover, each sequence is presented accompanied by its resolution.

- ***Caltrain.***

*Figure 3.23: Caltrain sequence (352×240) [227].*

- ***Carphone.***

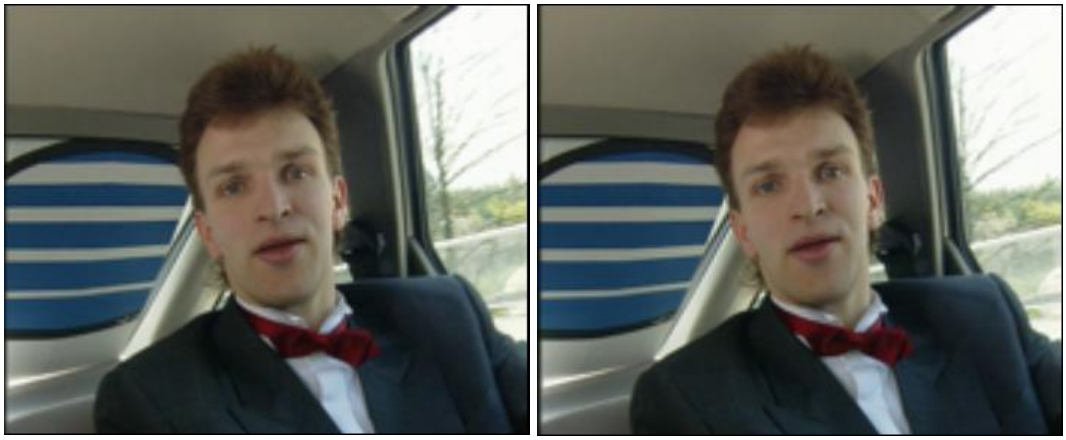

*Figure 3.24: Carphone sequence (176×144) [227].*

- ***Garden.***

*Figure 3.25: Garden sequence (352×240) [227].*

- ***Football.***

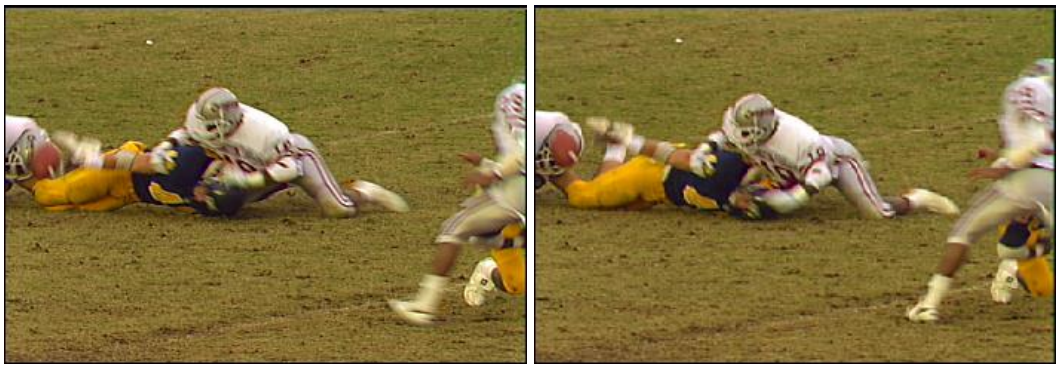

*Figure 3.26: Football sequence (352×240) [227].*

- ***Foreman.***

*Figure 3.27: Foreman sequence (176×144) [227].*

### 3.3.2. Error metrics.

There are several error metrics used along this work. Next, we are going to describe all of them.

- SAD (Sum Of Absolute Differences): this metric, previously shown at expression 3.1, has been used in every stage of this work when computing each one of the block-matching algorithms which have settled the basis for the different accelerations processes. This metric consist on a summation of every absolute difference between pixels values.

- MSE (Mean Squared Error): this metric is similar to the previously defined SAD, but using the squared subtractions of the pixels values for the summation instead of the absolute difference. MSE is characteristic for being less conservative than SAD, emphasizing the larger differences. Another difference with SAD is that the result of the summation is divided by the square of the macroblock size. Although it is not used directly in this work, it is used indirectly through PSNR metric as shown in formula 3.3:

$$MSE(x,y;u,v)=\frac{1}{N^2}\sum_{x=0}^{N-1}\sum_{y=0}^{N-1}\left|I_t(x,y)-I_{t-1}(x+u,y+v)\right|^2 \tag{3.3}$$

Where $I_t(x,y)$ is the pixel value in the coordinate $(x,\ y)$ in frame $t$, $(u,v)$ represents the motion of the candidate macroblock, and $N$ is the macroblock size.

- KPPS (Kilo Pixels Per Second): this metric is used for measuring the sensor throughput delivered in some stages of this work. It is very common when focusing on motion estimation techniques for measuring the throughput that could be achieved. This metric means the number of thousands of pixels that the device or algorithm which is being measured can process in one second.

- PSNR (Peak Signal to Noise Ratio): this metric is used for measuring the accuracy when using block-matching techniques. In some stages of this work this metric has been used for measuring the accuracy of the different block-matching algorithms using different inputs. Its formula is shown in expression 3.4:

$$PSNR(x,y;u,v) = 20\log_{10}\left[\frac{Max\_value}{\sqrt{MSE}}\right] \tag{3.4}$$

Where $Max\_value$ refers to the peat-to-peak value of the original data, which depends on the frame-grabber or the camera datasheet used. In the case of our work, it corresponds to an 8-bits range, so an intensity value of 256. This value characterizes the motion compensated image created by using motion vectors and macroblocks from de reference frame.

- Used resources: this is not purely a metric, but it means the used hardware resources which have been used along the different stages of this work. This metric has been used to evaluate the presented results in each case making a comparison. Inside this category, several kinds of resources have been established like reference, where between the most important we can find the number of logic elements, the number of embedded multipliers, the number of dedicated logic registers, the number of DSP (Digital Signal Processors), or the total number of memory bits used by the presented design.

- Time: the time has been the simplest and the most used metric during this work. It has been used for measuring the improvement achieved for every block-matching technique in every stage of the work using the millisecond as the main unit for the measures.

# Chapter IV

## Embedded systems

This chapter presents an introduction into the embedded systems family and its different types. Moreover, it is presented the SoPC design flow and the tool provided by Altera to design these systems.

Also, a detailed introduction to the Nios II processor is followed by a description of its different types. The different peripherals available around the Nios II embedded processor are shown accompanied by their descriptions.

As chapter closure, some future trends are shown.

## 4.1. Introduction.

In the computer world there are a lot of systems, which can be distributed in two main families.

On the one hand, we find the "general purpose computer system", which is a general computing platform and itself is the end product. General purpose computer systems are designed to be very flexible to support all the needs requested by end-users. Thus, application programs are developed based on the known available resources of this kind of systems. The main system in this family is the PC (Personal Computer), also known as desktop system.

On the other hand, we find the "embedded computer system", while not being in the majority of cases the end product, it is a computer system that performs one or few specific tasks. So, each embedded system answers to its own specifications. Because of its specific design, embedded computer systems are improved to develop their specific task reducing their cost and increasing their speed. To achieve these goals, this kind of computer systems has a resource constraint, so that they are only built with the just enough hardware resources for meeting their specific applications requirements [262][285].

Although computer systems can be divided in these two main groups, in the real world the division is not so easy, because many of computer systems are between these two families. If we think of a cell phone, we can catalog it in the "embedded computer systems family" because it is focused in wireless communication. But if we think now of a smartphone, it is designed for wireless communication, but also for loading video games, messaging applications ,or playing videos for example, which are characteristics nearer to a general purpose computer system.

Due to the fact that our focus is embedded systems, we are going to describe below their main distinguishing characteristics [262][286]:

- Cost: embedded systems must reduce the cost of the systems as much as possible. Although they are not low cost systems, embedded systems focus on cost reduction due to their specific design.

- General computation speed: although this is not the aim on an embedded computer system, some of them must perform general computation tasks with a good performance.

- Specific computation need: the key point in an embedded system is the performing of its specific tasks, which are mainly realized by customized hardware to improve speed.

- Real-time constraint: it means that the computation correctness not only depends on the final result, but also on the time this result is produced. Embedded computer real-time systems have always this characteristic, but not all the embedded systems are real time systems.

- Reliability: embedded computer systems must be reliable because they usually perform critical duties. So, when embedded systems are designed, all the possibilities must be taken into consideration to create a system which does not fail. For example, the embedded system used for flight control. If these systems failure, it will have disastrous consequences.

- Power consumption: embedded systems are designed taking into account their power consumption, although they are not low-cost systems. Because many embedded systems are portable systems, these are designed focusing on power consumption.

Once embedded computer systems main characteristics have been discussed, their main components are now being presented:

- Processor: the processor fetches the instructions from memory and executes them. These instructions are part of the processor instruction set and consist of an instruction code and some operands. Due to the fact that embedded processors are made for specific tasks, sometimes the processor is a specialized processor for the embedded computer system.

- Memory: in an embedded system, the memory usually consist of a ROM (Read Only Memory) type memory (only reading is allowed) and a RAM (Random Access Memory) type one (reading and writing are allowed). The memory in an embedded computer system has to have low access time and high density (more storage bits in the

same amount of space) because embedded systems do not usually have secondary storage devices.

- Peripherals: they are the output and input devices connected to the embedded computer system through ports. These ports can be of two types depending on the bits transferred between the peripheral and the embedded system at a time. Serial ports can only transfer one bit at a time, but parallel ports can transfer many bits at the same time.

- Hardware timers: they interrupt the processor at defined time periods for performing periodic tasks.

- Software: in embedded systems, the program code and the constant data are placed into the ROM, while the variables space is allocated in the RAM. In embedded systems the executed programs usually never end.

To finish this introduction to embedded systems, it is shown in Figure 4.1 the major application areas of embedded systems applications and the main embedded systems inside them [264].

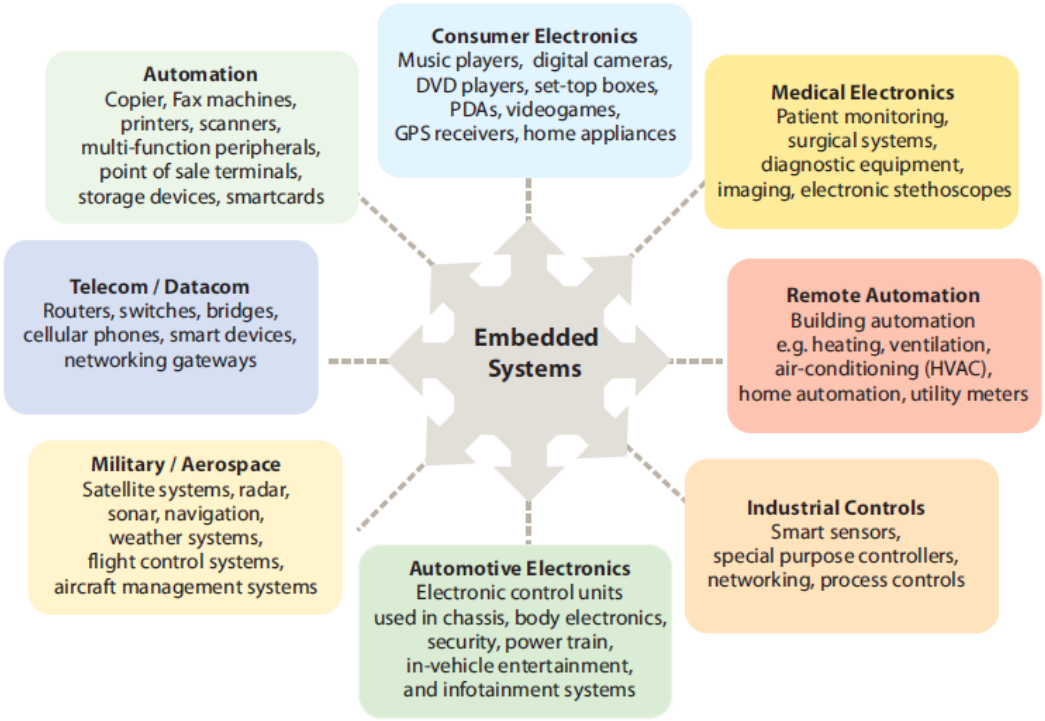


*Figure 4.1: Embedded systems devices and their application areas [264].*

## 4.2. History and state of art.

We have presented what an embedded computer system is, but we have not presented their history. To show the evolution of embedded systems we are going to describe its most famous milestones [265].

The seed of embedded systems was the integrated circuit designed in 1958 at Texas Instruments by Jack St Clair Kilby while Robert Noyce was also working separately on the invention. But it was in 1962 when Texas Instruments introduced the 7400 series of logic Integrated Circuits. Later, in 1965, Gordon Moore established his Moore's law and the 12 bit PDP-8 minicomputer was presented by DEC (Digital Equipment Corporation), considered the first computer embedded in instrumentation.

RCA released the first CMOS logic family used for high-end processing in 1968, while Fairchild presented the most popular op-amp circuit, the 741, which would be used in a great number of embedded systems. Two years later, in 1970, Texas Instruments developed a mask-programmable Integrated Circuit called PLA (Programmable Logic Array). Also in this year, Intel releases the first generally available DRAM (Dynamic Random Access Memory), called 1103. In 1971, the 555 timer was presented by Signetics while Intel released the first microprocessor which was a 4-bit CPU (Central Processing Unit), the 4004. Also in 1971, it was released the first EPROM (Erasable Programmable Read Only Memory).

In 1973, Texas Instruments introduced the first DRAM (Dynamic Random Access Memory). And three years later, in 1976, RCA released the first CMOS microprocessor. It was in 1978 when Intel presented the first x86 microprocessor, the 8086.

James Ready and Colin Hunter's company presented, in 1980, VRTX (Versatile Real Time Executive), the first commercial operating system for embedded systems. Also in this year, Altera released its first EPLD (Erasable Programmable Logic Device). But it wasn't until 1984 when the FPGA (Field Programmable Gate Array) was presented by some of the Xilinx co-founders. One year later, in 1985 Fujio Masuoka invents, while working with Toshiba, the flash memory.

In 1989, the first Embedded Systems Conference was held in San Francisco. Later, in 2004, Sony and IBM began to produce cell computer chips, although it was one year later when IBM, Intel, and AMD released the first multicore processor.

*Figure 4.2: First generation PDP-8 [265].*

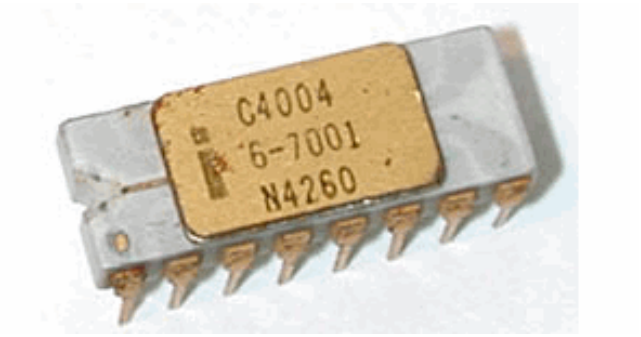


*Figure 4.3: The 4004, the chip which started the microprocessor revolution [265].*

Now, let us focus on embedded systems new trends [264][266]. Although there are a lot of new trends due to the fact that they form a growing computer system field, we are going to present only the most important ones:

- Multicore in embedded systems: due to the need of high performance, many embedded systems present multicore architecture, like smartphones or video gaming consoles. Even so, they have a big drawback which is their power consumption; although this issue is being tackled in the design of portable embedded systems.

- Embedded operating systems: current multicore embedded systems have to be served with multi-mission, multi-thread, multi-process, multiprocessor, and multi-board

debugging functionalities. Operating systems for embedded computer systems are being improved constantly.

- Embedded digital security systems: most of digital and surveillance security systems are embedded systems which use computer vision for capturing, post processing ,and identify the images or even objects inside the images.

- Embedded systems used in healthcare: the growth of biotechnology focused on healthcare is using a lot of embedded systems in sensors technologies, communications, and analytical problems. Also, the needs of diagnose imaging and monitoring make embedded systems more useful.

- Embedded systems inside automotive products: many embedded systems are used in the automotive industry, like ABS (Anti Block System) in the breaking subsystem. But nowadays, their presence has expanded into other parts inside automobiles like the engine control unit or fuel management for example.

- Embedded systems used for localization: in a global market, there is the need of being located to offer customized functionalities. Also in the entertainment field, the need of being located is very useful for social networks for example. These goals consume embedded systems for localization and internationalization.

### *4.2.1. Soft and hard processors.*

Processors can be cataloged into two main families attending to their core type when focusing on FPGA devices [270]. Although we are presenting both as follows, we are focusing on soft core processors due to the fact that they have been used during this work. The two main processors families attending to their core are:

- Hard-core processors: this kind of processors uses an embedded processor core mainly made of dedicated silicon plus the FPGA (Field Programmable Gate Array) basic logic elements. FPGAs that use hard-core processors are a hybrid approach between an ASIC (Application Specific Integrated Circuit) and a purer FPGA. As their best advantage, they usually present smallest area, higher clock rates, and higher performance than soft core processors. There are a lot of manufacturers which are

currently making hard-core processors, but the main manufacturers and devices holding these processors in the FPGA world, are shown below:

o Xilinx PowerPC 405: it [290] is a hard 32-bit processor with the PowerPC Harvard RISC (Reduced Instruction Set Computing) architecture. It is used in FPGA Virtex-II Pro with version 405D5, and also in FPGA Virtex-4 and Virtex-5 using its version 405F6. Here, its main features are described:

- Clock rate of 450 MHz.
- It achieves 700+ DMIPS (Dhrystone Millions Instruction Per Second).
- Five stages pipeline (Fetch, Decode, Execute, Memory, Writeback).
- Fixed-point execution unit including 32 registers of 32 bits.
- Hardware multiplier module.
- Hardware divider module.
- Data and instruction caches.
- MMU (Memory Management Unit).
- TLB (Translation Lookaside Buffer) with 64 unified inputs.
- Variable page size (1KB - 6 KB).
- Debugging through JTAG (Join Test Action Group) interface.

o ARM (Advanced RISC Machine) dual core Cortex-A9 MPCore: ARM Cortex A9 MPCore is a 32-bit RISC architecture developed by ARM Holdings. ARM architecture is the most widely used 32-bit instruction set. As a highlight of this architecture, the four higher bits of each instruction are used as conditional code, allowing conditional execution. Moreover, this architecture presents the possibility to add shifts and rotations in the data processing. Dual core Cortex-A9 MPCore, shown in

Figure 4.4, is used in Altera's Arria V [267] and Altera's Cyclone V [292]. Also, it is used in Xilinx Zinq-7000. Here, ARM Cortex-A9 MPCore [291] main features are described:

- Clock rate of 800 MHz on each core.
- It achieves 2.5 MIPS per MHz on each core.
- Single- and double-precision floating-point unit on each core.
- Data and instruction caches of 32 KB on each core.
- MMU (Memory Management Unit) on each core.
- Generic Interrupt Controller.
- 32-bit general purpose timer.
- Watchdog timer.

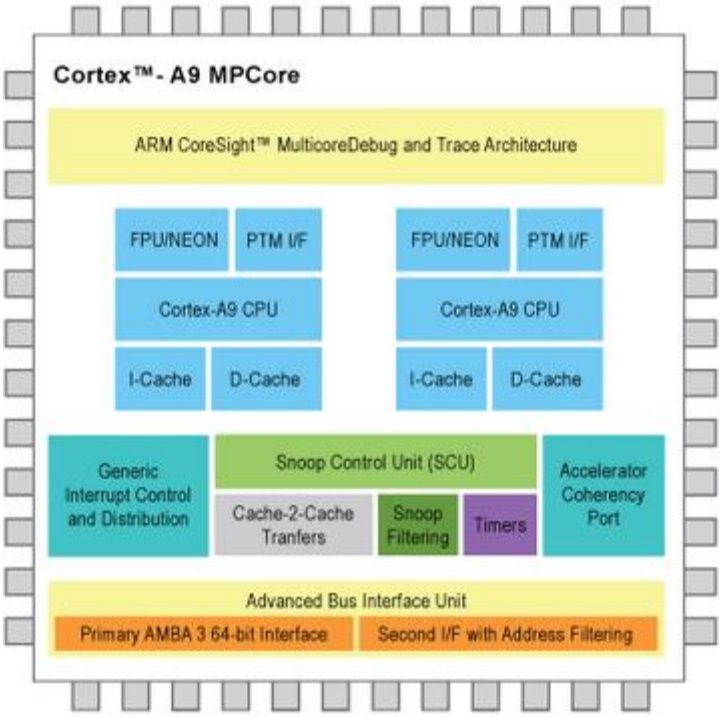


*Figure 4.4: Cortex-A9 MPCore block diagram [291].*

- Soft-core processors: they are built using the FPGA's programmable logic elements and are mainly described using a HDL (Hardware Description Language) or a netlist. So, opposite to hard-core processors, they are synthetized and fitted into the FPGA fabric. Due to this, they are very flexible and suitable for customization, allowing the designer to set the ALU (Arithmetic Logic Unit) or the memory address space among other features in compiling time. As drawbacks, soft-core processors present lower clock rates and consume more power than equivalent hard-core processors. As advantages, soft-core processors can be redesigned, they provide future protection against microcontroller variants, they are low-cost in system-level integration, and they can be exactly fitted for combining the microcontroller with peripherals. As in hard-core processors, there are a lot of manufacturers, but the main manufacturers and soft-core processors are listed below [269]:

    - Altera Nios II, which is used in both FPGAs and ASICs [273].
    - Xilinx MicroBlaze, which is used in both FPGAs and ASICs [284].
    - OpenRISC, which is an open source soft-core processor [294].
    - Leon4, which is an open source soft-core processor [293].

In Table 4.1, it is presented a simple comparison between the above presented soft-core processors focusing on the following features:

- Open source: it points is the soft-core processor is open source or private.
- Hardware FPU (Floating Point Unit): it points whether the processor has a FPU.
- Bus standard: it points the type of bus used in the soft-core processor.
- Integer division unit: it points if the processor has an Integer Division Unit.
- Custom coprocessors/instructions: it points whether custom instructions can be designed for the soft-core processor.
- Maximum frequency on FPGA (MHz): it presents the maximum clock rate frequency expressed in MHz which the processor could work.

- Max MIPS (Million Instructions Per Seconds) on FPGA: it presents the maximum MIPS shown on marketing material, so it could vary.

- Resources: it shows the number of the FPGA basic resources needed for the soft-core processor accomplishment.

- Area estimate: it shows the estimated FPGA area which the soft-core processor will fill.

| | Nios II (fast) | MicroBlaze | OpenRISC | Leon4 |
|---|---|---|---|---|
| Open source | No | No | Yes | Yes |
| Hardware FPU | Yes | Yes | No | Yes |
| Bus standard | Avalon | CoreConnect | WISHBONE | AMBA |
| Integer division unit | Yes | Yes | No | Yes |
| Custom coprocessors/ instructions | Yes | Yes | Yes | Yes |
| Maximum frequency on FPGA (MHz) | 290 | 200 | 47 | 125 |
| Max MIPS on FPGA | 340 | 280 | 47 | 210 |
| Resources | 1800 LE | 1650 slices | 2900 slices | 4000 slices |
| Area estimate | 1500 $A$ | 800 $A$ | 1400 $A$ | 1900 $A$ |

*Table 4.1: Soft core processors [269].*

## 4.3. SoPC design flow and comparisons.

Embedded computer systems are usually accompanied by Input/Output peripherals, to provide an interface for the user and for special hardware accelerators, helping them manage high cost computations. All these components could be taken into a single integrated circuit, which is known as SoC (System on Chip). If we translate this design methodology to an FPGA chip, we would be in front of a SoPC (System on a Programmable Chip) also known as PSoC (Programmable System on a Chip) [288].

The designed hardware features are usually described in a HDL (Hardware Description Language) to program the FPGA device interconnecting its logic cells. Due to this easy hardware programming, embedded computer systems could incorporate many times this customized hardware. A SoPC based embedded system gives a more flexible hardware design and makes it easier to customize software for the needed requirements.

Once the embedded SoPC computer system bases have been established, we are presenting below the four flow stages to develop an embedded SoPC system, using the Altera Nios II soft-core processor as a processor example:

- Hardware/software partition: due to in a SoPC system tasks can be implemented in hardware, software, or both, we have to decide which task will be implemented through hardware, software, or both [288]. We have to take into account the available hardware resources and the performance requirements for taking any decision.

- Hardware development: it can be divided into the following steps:

  o Nios II processor type: from the available soft cores provided by Altera, we can choose the most fitting to our SoPC system.

  o I/O standard peripherals: we will select the needed peripherals from the ones provided by Altera.

  o HA (Hardware Accelerators): these must be designed from scratch to perform specific tasks, which we want to achieve high throughput.

  o User I/O peripherals: maybe there are not standard peripherals for all the needs of our SoPC system, then we must design them from scratch like hardware accelerators.

  o User logic: this is the hardware logic added to the designed SoPC system which does not interact directly with the Nios II processor.

- Software development: the software developed for a SoPC system is not all user developed software [2.88]. Much software is provided by Altera, integrated into its HAL (Hardware Abstraction Layer) for providing I/O peripherals communication among other functionalities. Even so, we can catalog the software developed in a SoPC system into API (Application Programming Interface) functions (which are into the Altera HAL), user I/O drivers (developed for custom I/O peripherals or hardware accelerators communication), and user functions (the base of the application program).

- Physical implementation and testing: this step consists in downloading the designed device to the FPGA, for later loading the developed software into the ROM memory [288]. After this, the SoPC system can be tested.

In Figure 4.5, we show the SoPC development flow using an Altera FPGA Nios II based. The hardware part development for the SoPC system is shown on the left while the software development part is shown on the right.

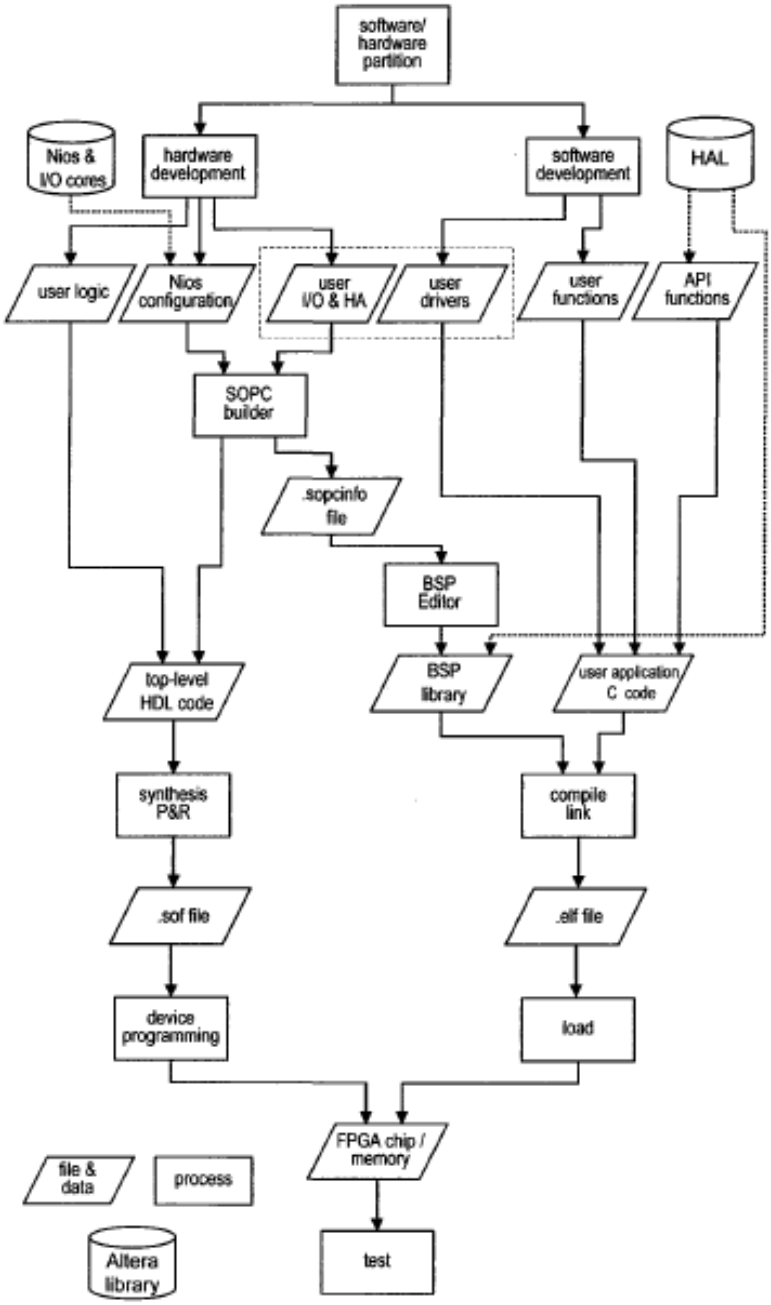


*Figure 4.5: SoPC design flow [263].*

## 4.4. Altera SOPC Builder.

Altera SOPC builder completes the hardware achievement during the design and development of a SoPC system. This program allows us to define and generate from scratch a complete SoPC much faster than doing it manually.

To design a SoPC system, SOPC Builder offers a GUI (Graphic User Interface) to interact between the hardware components [271]. Later, it will generate HDL files for connecting every hardware component as designed with the GUI. Also, SOPC Builder allows HDL files developed by the user. But these files must be Verilog and/or VHDL files.

SOPC Builder builds through its GUI a SoPC system which is composed of different components. These components can be chosen from the ones provided by Altera, or can be designed and developed by the user. Joining all the components in the system, SOPC Builder creates a top level HDL file which defines the whole SoPC system, and the interconnections between all the components of the system [271].

A SoPC system component can be defined as the instance of a module. Every SOPC Builder module uses an Avalon interface; such as memory mapped, streaming, or IRQ (Interruption Request) for interconnecting with other modules in the system.

Looking inside a design developed with SOPC Builder, a component can be a logical device contained completely into the SOPC Builder system, or it can work like an interface for an off-chip device, for example a DDR2 memory controller in Figure 4.6.

In addition to the Avalon interface, there can be other types of signals like PIO (Peripheral Input Output) which are used to provide connection to the logic outside the SoPC built system.

These kinds of interconnections between components are shown in Figure 4.6.

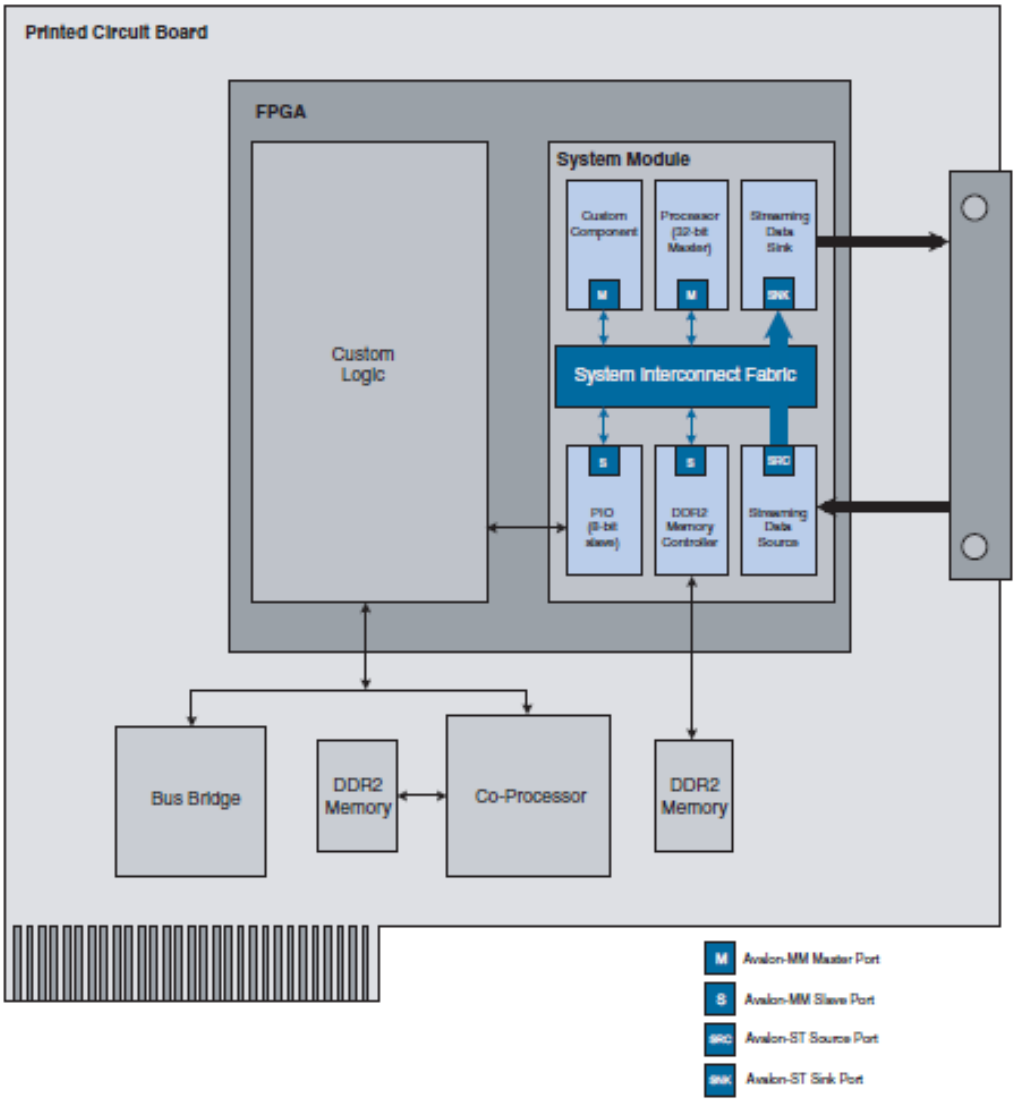


*Figure 4.6: Example of system generated by Altera SOPC Builder [271].*

Available modules in Altera SOPC Builder can be cataloged into three different types:

- Ready to use components: these components presented in the SOPC Builder are made by Altera or third party providers, and are ready to use without having to perform any action, more than add them to the system. In this component type, we can find microprocessors like Nios II, timers, UARTs (Universal Asynchronous Receiver Transmitter), or interfaces to off-chip devices among others.

- Custom components: because the user can define and develop hardware modules through HDL files using VHDL or Verilog programming languages, the SOPC Builder allows the user to add custom components to the system.

- Third Party components: Altera provides different IP (Intellectual Property) functions made by other providers but ready to use in the SOPC Builder.

Insomuch as SOPC Builder has many tools, its main functions are going to be described as well as its main flow for design. With SOPC Builder, the user can design the hardware design structure through its GUI. This useful GUI allows you to add, remove, or alter every component as well as the connections between them in the designed system.

To design a hardware system using SOPC Builder it is useful to follow the provided design flow [271]. This design flow is divided into several main steps which are described as follows:

- Use the component editor for packing our own components.

- Simulation at the unit level for ensuring the correct working of the components.

- Build the system adding the needed components specifying clocks or addresses among other features.

- Generate the system through the SOPC Builder generation tool. In this step, SOPC Builder will parameterize all the components as asked, will create the different interconnections between the components, and finally will generate the system through HDL files. These files are:

    - A HDL file for the top level system.

    - A HDL file for each component in the system.

    - A SDC (Synopsis Design Constraint) file for timing analysis.

    - A BSF (Block Symbol File) representing the top level system.

    - A data sheet.

o A SOPC information file with a fine grain description of the whole system

- Simulation at the system level.
- Develop and compile the software for the design.
- Download the complete system to the FPGA device.
- Test in hardware.

In Figure 4.7, we show a synthesis of the design process of a SoPC system using Altera SOPC Builder.

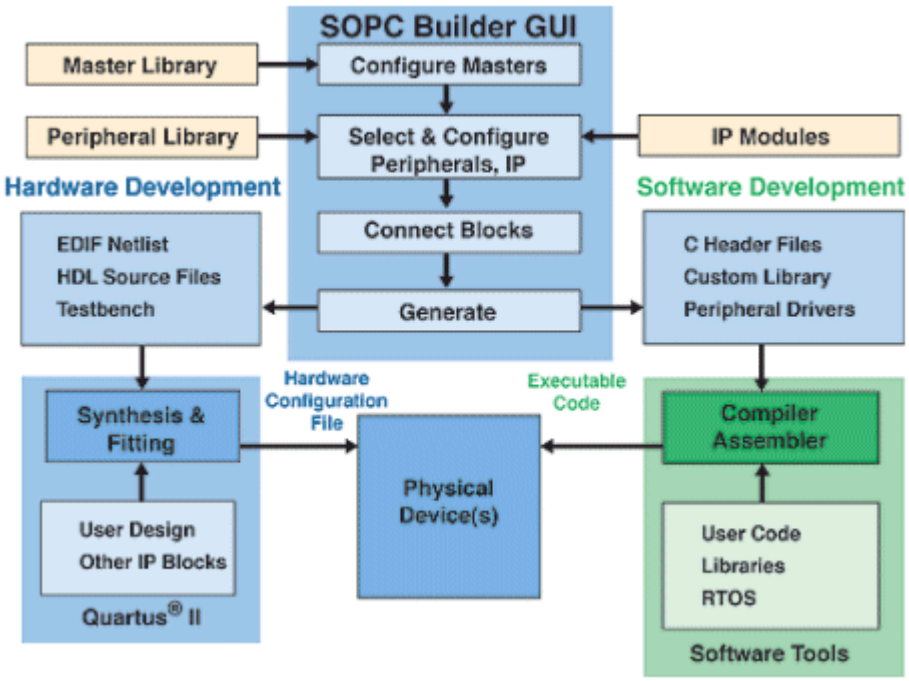


*Figure 4.7: SoPC design flow using Altera SOPC Builder [272].*

## 4.5. Nios II and low-cost boards DE2-C35 and DE2-C115.

The Altera Nios II processor is a soft-core RISC (Reduced Instruction Set Computing) processor [273]. It is composed of the Nios II processor core, a group of on-chip peripherals, an on-chip memory, and interfaces for off-chip memories. Due to the fact that the Nios II processor is a soft-core processor, it is possible to add or remove features or peripherals. Even the address map can be configured in the Nios II processor

system. This address map can be divided into reset address, exception address, and break handler address sections.

The Nios II processor core is characterized by the following features:

- 32-bit instruction set, datapath, and address space.
- 32-bit registers.
- 32 interrupt sources.
- 32×32 singles instructions for multiply and divide.
- Floating point instructions.
- Access to on-chip peripherals.
- Interfaces to off-chip memories and peripherals.
- Hardware assisted debug mode.
- Optional MMU (Memory Management Unit).
- Optional MPU (Memory Protection Unit).
- Performance up to 250 DMIPS (Dhrystone Millions Instruction Per Second).
- Custom instructions.

Once reviewed Nios II core features, now it is presented the hardware structure of this Nios II processor core [273]. The Nios II hardware architecture is composed of the Nios II processor core and the peripherals connected to this core. The Nios II core architecture is composed of the following functional units:

- Program Controller & Address Generation: this module provides generation of the different addresses where the program instructions are read, and executed program controlled, through several internal components apart from three different signals for resetting and debugging purposes. One of these signals forces the processor core to globally reset immediately, other reset locally only the processor core but not other

components in the Nios II system, and the last suspends the processor core execution for debugging purposes.

- Register file: Nios II processor core register file consists on thirty two 32-bit general purpose integer registers, up to thirty two 32-bit control registers, and up to sixty three shadow register sets. These shadow register sets consist of thirty two 32-bit registers and are mainly used for context switching. Moreover, the Nios II architecture allows future floating point registers addition.

- ALU (Arithmetic Logic Unit): it operates with the data stored in one or two general purpose registers, depending on the input, and store the result back in a general purpose register. It supports different operations which can be divided into four main families:

    o Arithmetic: addition, subtraction, multiplication, and division on signed and unsigned operands.

    o Relational: equal, not equal, greater than or equal, and less than on signed and unsigned operands.

    o Logical: AND, OR, NOR and XOR.

    o Shift and Rotate: shift and rotate data by 0 to 31 bit positions.

Because the Nios II architecture admits user defined custom instructions, the ALU is connected directly to custom instruction logic, for accessing to the hardware implementing the custom instructions and using it like the native instructions.

- Exception and interrupt controllers: the Nios II processor core includes hardware components for exception handling. These components are three:

    o Exception Controller: it is a simple exception controller for handling all exception types.

    o EIC (External Interrupt Controller): it provides high performance hardware interrupt interface to reduce interrupt latency through an external interrupt controller.

o IIC (Internal Interrupt Controller): it provides management for internal hardware interrupts.

- Instruction and Data buses: the Nios II hardware architecture provides different buses for instruction and data. Both are implemented as Avalon-MM master ports. The instruction master port connects to memory components while the data master one connects to both memory and peripherals.

- Instruction and Data caches: they are situated in the instruction and data master ports, residing on chip, improving the memory access time for off-chip memories.

- Tightly Coupled Memory interfaces: they provide interfaces to tightly coupled memories outside the Nios II processor core. These memories are placed on chip and are similar to caches but without real-time caching overhead. They can be used for both, instruction and data access.

- MMU (Memory Management Unit): it is optional, but when used, it provides 32-bit virtual to physical address mapping. It uses hardware TLBs (Translation Lookaside Buffer) to improve the speed in the virtual address translation process. It is mutually exclusive from the MPU component.

- MPU (Memory Protection Unit): it is optional, but when used, it provides memory protection for up to 32 variable size instruction regions and up to 32 variable size data regions. It also manages writing and reading permission for data regions. It is mutually exclusive from the MMU component.

- JTAG debug module: it provides features for debugging remotely the Nios II processor core from a host PC. It is used for downloading programs to memory, starting/stopping the execution, or setting breakpoints/watchpoints between other tasks. Its main drawback is that it is not supported by the MMU (Memory Management Unit).

Below, we show the Nios II processor core hardware architecture as described before, and a zoom into this architecture focusing on instruction and data possibilities.

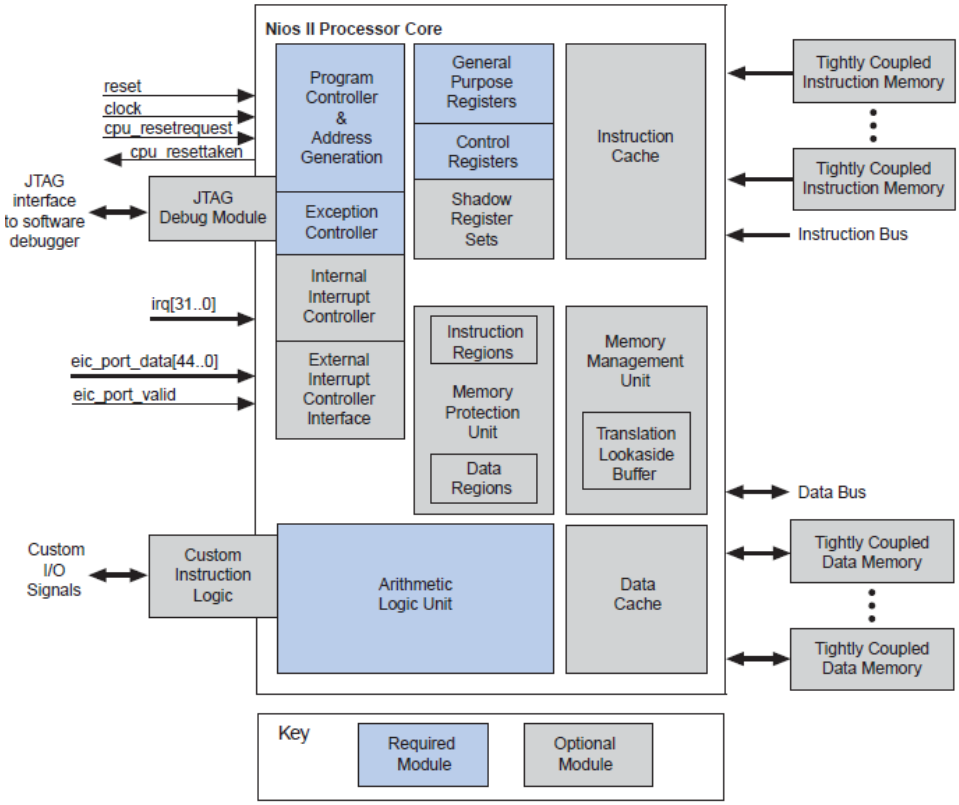


*Figure 4.8: Nios II processor core hardware architecture [273].*

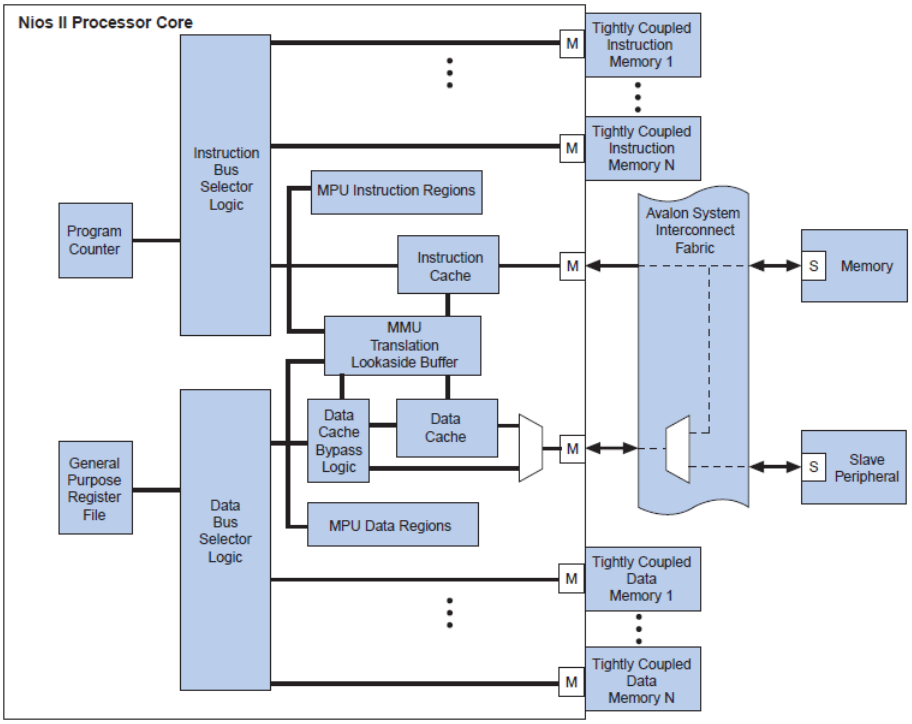


*Figure 4.9: Nios II memory and Input/Output block diagram [273].*

A general description of the Nios II processor core has been presented above. Now that one of the key features of Altera FPGAs has been shown, we are going to describe the two main low cost FPGAs [274][276].

In the one hand, we present the DE2-C35, which is an FPGA board designed for learning purposes under low cost constraints [275]. It consists of a Cyclone II EP2C35 (672 pins) FPGA connected to several peripherals. A real image of the DE2-C35 is shown in Figure 4.10.

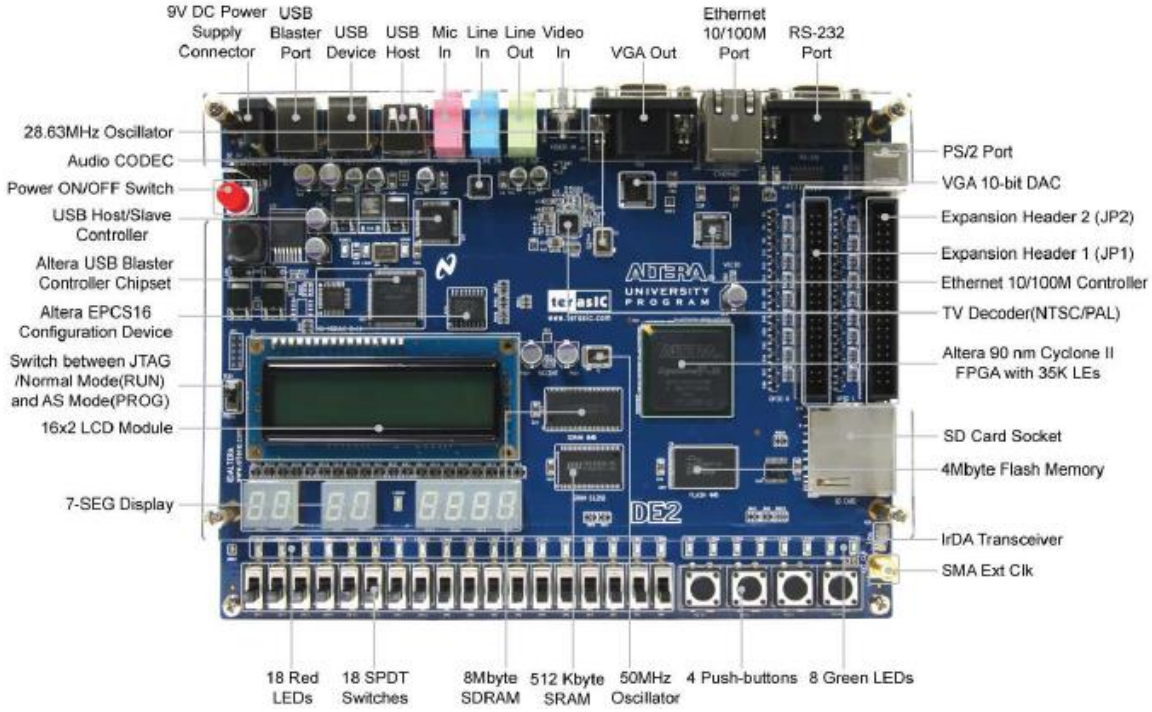


*Figure 4.10: DE2-C35 board [275].*

Now, a block diagram which shows the different features of the DE2-C35 in a friendly way is presented in Figure 4.11.

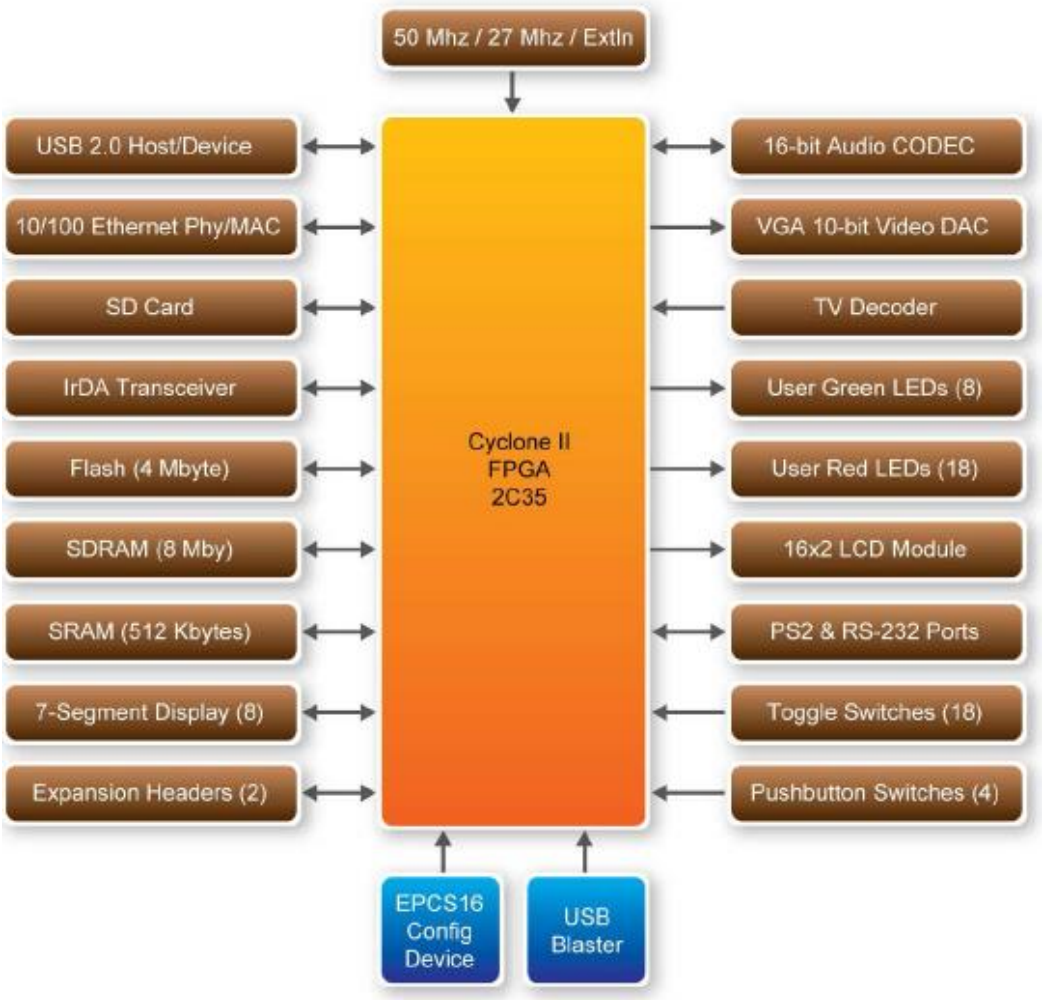


*Figure 4.11: DE2-C35 board block diagram [275].*

On the other hand, we present the DE2-C115 [277], which is an FPGA board designed for learning purposes under low-cost constraints, but with higher throughput than the DE2-C35 and the DE2-C70, an intermediate solution between the presented FPGA boards. It consists of a Cyclone IV EP4CE115 (780 pins) FPGA connected to several peripherals. A real image of the DE2-C35 is shown in Figure 4.12.

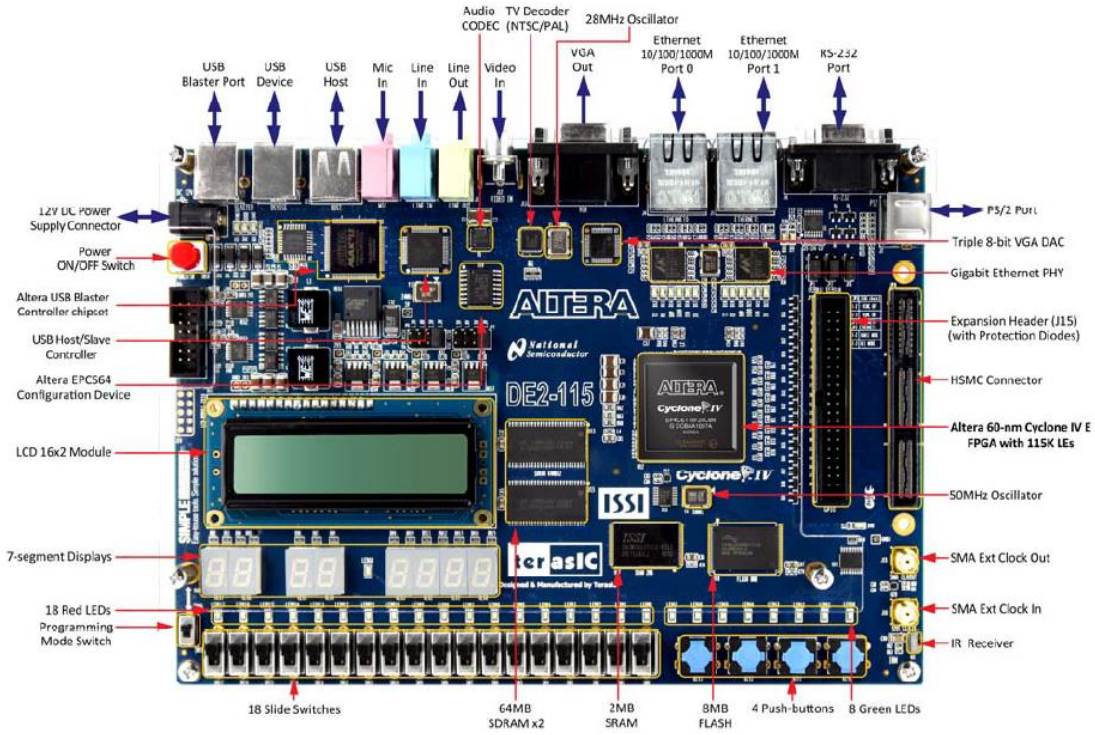


*Figure 4.12: DE2-115 board [277].*

In Figure 4.13, a block diagram which shows the different peripherals connected to the Cyclone IV integrating the DE2-C115 board is presented in a friendly way.

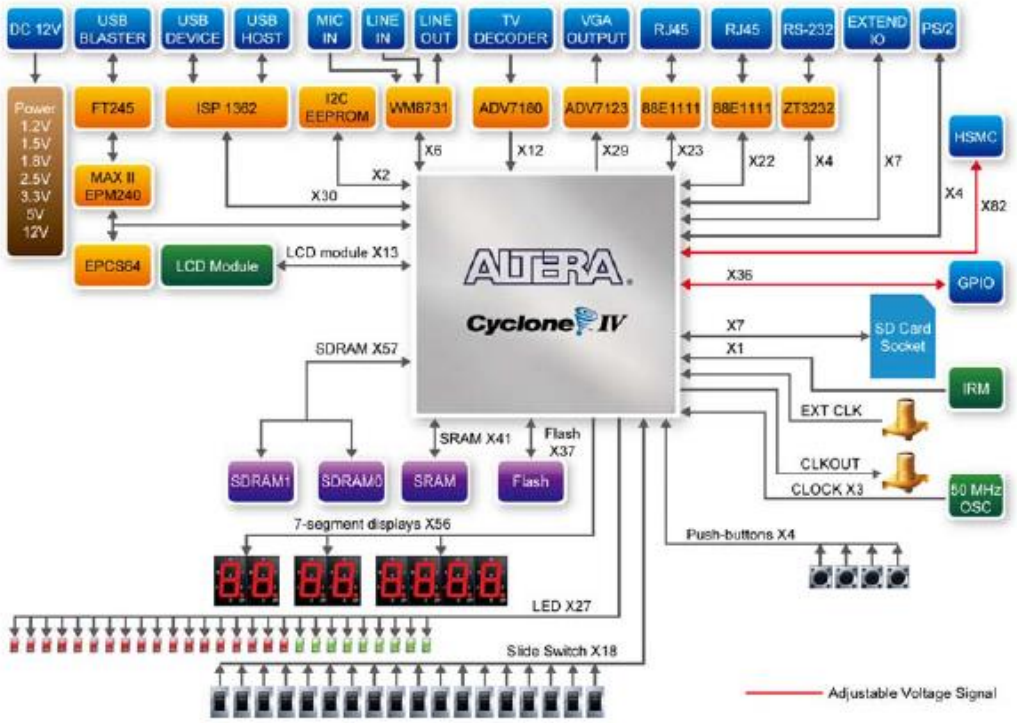


*Figure 4.13: DE2-C115 board block diagram [277].*

### *4.5.1. Nios II processor core types.*

The Nios II processor core allows three different configurations [273]. In advance, they are presented using a brief description and a features table, but later, they will be described in detail. Each one of the Nios II processor cores is optimized for one target, being them:

- Nios II/e (economic): this core is designed for low cost, achieving the smallest possible core size. Because of this, it has limited features, limiting many settings available in other Nios II processor core configurations [273].

- Nios II/s (standard): this core is designed for low cost but maintaining a certain grade of performance. Due to this, it has not got all the possible features in the fast core configuration [273].

- Nios II/f (fast): this core is designed for throughput purpose. It achieves the fastest performance among the different core configurations. Moreover, it allows tuning the processor through the most configuration options for achieving a higher performance [273].

In Table 4.2, we show the main differences between the three available Nios II processor core default configurations.

| Feature | | Core | | |
|---|---|---|---|---|
| | | **Nios II/e** | **Nios II/s** | **Nios II/f** |
| Objective | | Minimal core size | Small core size | Fast execution speed |
| Performance | DMIPS/MHz | 0.15 | 0.74 | 1.16 |
| | Max. DMIPS | 31 | 127 | 218 |
| | Max. $f_{MAX}$ | 200 MHz | 165 MHz | 185 MHz |
| Area | | < 700 LEs;<br>< 350 ALMs | < 1400 LEs;<br>< 700 ALMs | Without MMU or MPU:<br>< 1800 LEs;<br>< 900 ALMs<br>With MMU:<br>< 3000 LEs;<br>< 1500 ALMs<br>With MPU:<br>< 2400 LEs;<br>< 1200 ALMs |
| Pipeline | | 1 stage | 5 stages | 6 stages |
| External Address Space | | 2 GB | 2 GB | 2 GB without MMU<br>4 GB with MMU |
| Instruction Bus | Cache | – | 512 bytes to 64 KB | 512 bytes to 64 KB |
| | Pipelined Memory Access | – | Yes | Yes |
| | Branch Prediction | – | Static | Dynamic |
| | Tightly-Coupled Memory | – | Optional | Optional |
| Data Bus | Cache | – | – | 512 bytes to 64 KB |
| | Pipelined Memory Access | – | – | – |
| | Cache Bypass Methods | – | – | ■ I/O instructions<br>■ Bit-31 cache bypass<br>■ Optional MMU |
| | Tightly-Coupled Memory | – | – | Optional |
| Arithmetic Logic Unit | Hardware Multiply | – | 3-cycle | 1-cycle |
| | Hardware Divide | – | Optional | Optional |
| | Shifter | 1 cycle-per-bit | 3-cycle shift | 1-cycle barrel shifter |
| JTAG Debug Module | JTAG interface, run control, software breakpoints | Optional | Optional | Optional |
| | Hardware Breakpoints | – | Optional | Optional |
| | Off-Chip Trace Buffer | – | Optional | Optional |
| Memory Management Unit | | – | – | Optional |
| Memory Protection Unit | | – | – | Optional |
| Exception Handling | Exception Types | Software trap, unimplemented instruction, illegal instruction, hardware interrupt | Software trap, unimplemented instruction, illegal instruction, hardware interrupt | Software trap, unimplemented instruction, illegal instruction, supervisor-only instruction, supervisor-only instruction address, supervisor-only data address, misaligned destination address, misaligned data address, division error, fast TLB miss, double TLB miss, TLB permission violation, MPU region violation, internal hardware interrupt, external hardware interrupt, nonmaskable interrupt |
| | Integrated Interrupt Controller | Yes | Yes | Yes |
| | External Interrupt Controller Interface | No | No | Optional |
| Shadow Register Sets | | No | No | Optional, up to 63 |
| User Mode Support | | No; Permanently in supervisor mode | No; Permanently in supervisor mode | Yes; When MMU or MPU present |
| Custom Instruction Support | | Yes | Yes | Yes |

*Table 4.2: Nios II processor core types and their features [273].*

- ***Nios II/e.***

The Nios II/e processor core is designed for minimizing the use of hardware resources, maintaining the compatibility with the Nios II processor instruction set architecture [273]. Indeed, its size is half the Nios II/s processor core size. Its main drawback is that it presents a lower throughput compared with the other Nios II processor core configurations, but it is designed for low-cost applications. Its main features are described below:

- Executes one instruction in at least six clock cycles.

- Custom instructions are addable.

- JTAG debug module is available but does not support hardware breakpoints.

- Software emulation of unimplemented instructions (does not provide hardware support).

- Access up to 2 GB of external address space.

- No instruction/data cache (every memory/peripheral access generates an Avalon-MM transfer).

- No branch prediction due to a one stage pipeline.

- Dedicated shift circuitry (one bit per cycle) for shift and rotate.

- Support for exception handling (illegal/unimplemented instruction, internal hardware interrupt, and software exception).

- ***Nios II/s.***

The Nios II/s processor core is also designed for minimizing hardware resources, but improving execution throughput [273]. Its execution throughput is around 40% less than Nios II/f processor core but used hardware resources are 20% of those used by Nios II/f. This Nios II processor core is designed for medium cost and performance applications. Its main features are described below:

- Executes one instruction in one clock cycle (normal ALU instructions) or more.

- Instruction direct mapped implementation cache available with user defined size between 512 bytes and 64KB, reading an entire cache line at a time.

- Access up to 2 GB of external address space.

- Custom instructions are addable.

- No data cache.

- Five stages pipeline (Fetch, Decode, Execute, Memory (it only can create stalls), Writeback).

- Static branch prediction using branch offset direction (negative for backward branch and positive for forward branch).

- Provides hardware multiply options (DSP multipliers, embedded multipliers, and multipliers built from LE (Logic Elements)), divide options, and shift options (three or four clocks cycles when a hardware multiplier is present, or one bit per cycle through dedicated circuitry).

- JTAG debug module is available including hardware breakpoints and real-time trace.

- Support for exception handling (illegal/unimplemented instruction, internal hardware interrupt, and software exception).

- Support for up to four tightly coupled memories for instructions, bypassing instruction cache memory, using each one a memory interface master port and connected directly to one memory slave port.

- Data master port is always present, but instruction master port is included when instruction cache is present.

- ***Nios II/f.***

The Nios II/f processor core is designed for high execution performance through an expensive core size [273]. If we compare its size with the Nios II/s size, it is nearly a

25% bigger. As the highest performing Nios II processor core, it is designed for performance critical applications. Its main features are described below:

- Custom instructions are addable.

- Access up to 2 GB of external address space when MMU is not present and up to 4 GB when MMU is present

- Up to 63 optional shadow register sets.

- Dynamic branch prediction using a two bit branch history table.

- Provides hardware multiply options (DSP multipliers, embedded multipliers, and multipliers built from LE (Logic Elements)), divide options, and shift options (one or two clocks cycles when hardware multiplier is present, or one bit per cycle through dedicated circuitry).

- JTAG debug module is available including hardware breakpoints and real-time trace except for MMU.

- Optional instruction (instruction cache enabled) and data (data cache enabled) master ports. Bursting is allowed when data cache is enabled.

- Executes one instruction in one clock cycle (most ALU instructions) or more.

- Instruction direct mapped implementation cache available with user defined size between 512 bytes and 64KB, reading an entire cache line (32 bytes) at a time in one clock cycle. It is virtually indexed when MMU is present.

- Data direct-mapped implementation cache available with user defined size between 512 bytes and 64KB, reading an entire cache line (configurable of 4, 16, or 32 bytes) at a time in one clock cycle. It is virtually indexed when MMU is present.

- Support for both instructions and data tightly coupled memories, bypassing instruction and data cache memories when enabled, using each one a memory interface master port and connected directly to one memory slave port. When MMU is present

tightly coupled memories are mapped into the kernel partition being only accessed in supervisor mode.

- Six stages pipeline (Fetch, Decode, Execute, Memory, Align -only it and Decode stage can create stalls-, Writeback).

- Optional MMU providing a TLB (Translation Lookaside Buffer), which consists on a main TLB, a micro TLB for instructions (µITLB), and another micro TLB for data (µDTLB), both stored using LE based registers. Both µTLB are used as cache for the main TLB and are not visible to software.

- Optional MPU.

- Support for exception handling (illegal/unimplemented instruction, internal hardware interrupt, software exception, misalignment, division error, TLB exception, MPU region violation, and supervisor only instruction/instruction address/data address).

- Optional EIC (External Interrupt Controller) interface to boost interrupt handling through several signals (RHA (Requested Handler Address), RRS (Requested Register Set), RIL (Requested Interrupt Level), and RNMI (Requested Nonmaskeable Interrupt)).

### *4.5.2. Peripherals.*

Before turning to peripherals, we will explain that an IP (Intellectual Property) core is like a logic unit, cell, or reusable design, which has a specific task to develop. An IP core can be property of a group or a single person, and they are used as design blocks for ASICs chips or logic designs on FPGAs [290].

Focusing on the design of SoPC systems through the Altera Nios II development environment, we have several IP cores available provided by Altera [278 – 280]. These cores, working like peripherals in an embedded system, can be quickly instantiated using the Altera SOPC Builder tool, providing software driver support for the Nios II processor. Moreover, these cores can be used in any system loaded in an Altera FPGA.

Here we present the most commonly used peripherals and the controllers provided by Altera:

- SDRAM (Synchronous Dynamic Random Access Memory) controller: it allows the designer to connect in an easy way to SDRAM chips thanks to the Avalon interface. This controller [278 – 279] supports PC100 standard SDRAM chips and is able to connect to several SDRAM chips which run at the same clock rate as the Avalon interface. The connection between the Nios II core and the external SDRAM chip is done using an Avalon slave port, which supports wait states for read and write transfers, connected to an Avalon master port. This core is able to access SDRAM subsystems with several memory sizes and several data widths (8, 16, 32, or 64). Because of the using of the Avalon interface, read transfers are pipelined. Moreover, this controller can share its data buses and address with other off-chip Avalon devices allowing more peripherals in systems with limited I/O pins.

In Figure 4.16 we show a connection example between the embedded designed system and an external 128 Mbits SDRAM chip with 32-bit data.

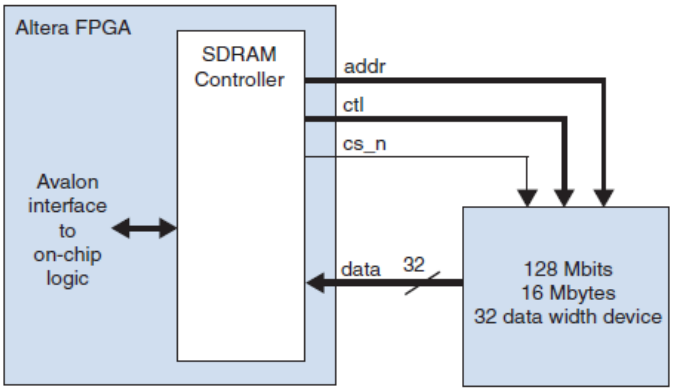


*Figure 4.14: Example connection to external SDRAM memory chip [278].*

- CFI (Common Flash Interface) controller: it [278 – 280] allows the designer to connect in an easy way to flash memory chips thanks to the Avalon interface. Altera provides for the Nios II processor driver routines through the HAL (Hardware Abstraction Layer) [278 – 280] so that it is not needed to write extra code for managing the external flash memories. Moreover, it is provided a flash programmer. The connection between the Nios II core and the external flash memory chip is done using an Avalon tristate bridge which allows the flash memory chip to share its data buses and address with other memory chips.

In Figure 4.15 we present a connection example for an external flash memory.

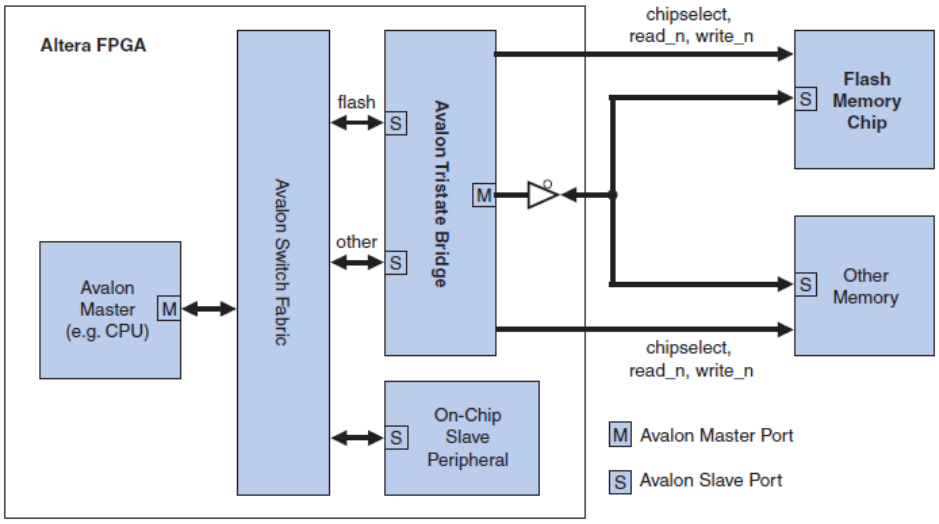


*Figure 4.15: Example connection to external Flash memory chip [278].*

- EPCS (Erasable Programmable Configurable Serial) controller: the EPCS device is just a rebranded SPI (Serial Peripheral Interface) flash device with minor changes which can be mainly separated into two main regions: a FPGA configuration memory, and a General Purpose memory made of the rest of space in the EPCS device. Once defined the EPCS device, we are going to present the EPCS controller [278 – 281].

The EPCS controller allows the designer to connect in an easy way to the serial EPCS flash memory. Also in this controller, Altera provides drivers integrated into the HAL system library [278 – 280] to read and write data into the EPCS device. Using this controller, Nios II systems are able to store program code, nonvolatile program data (e.g. a serial number among other persistent data), and/or FPGA configuration in the EPCS device. Also, Altera provides a flash programmer utility to manage data into the EPCS device.

In Figure 4.16, we show an example interconnection between an EPCS device and a system based on Nios II through the Avalon switch Fabric.

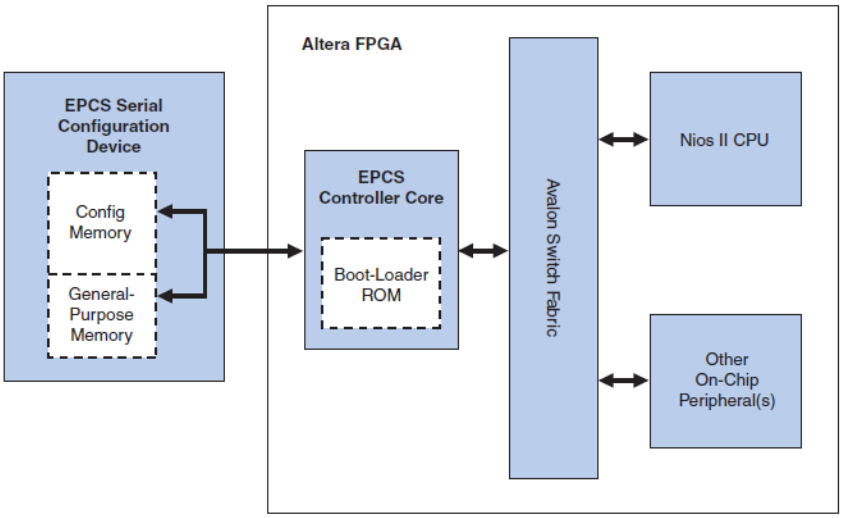


*Figure 4.16: Example connection to external EPCS memory chip [278].*

- UART (Universal Asynchronous Receiver/Transmitter): its function [278 – 280] is to provide a communication between an external device and the embedded system implemented in the Altera FPGA using the RS-232 protocol.

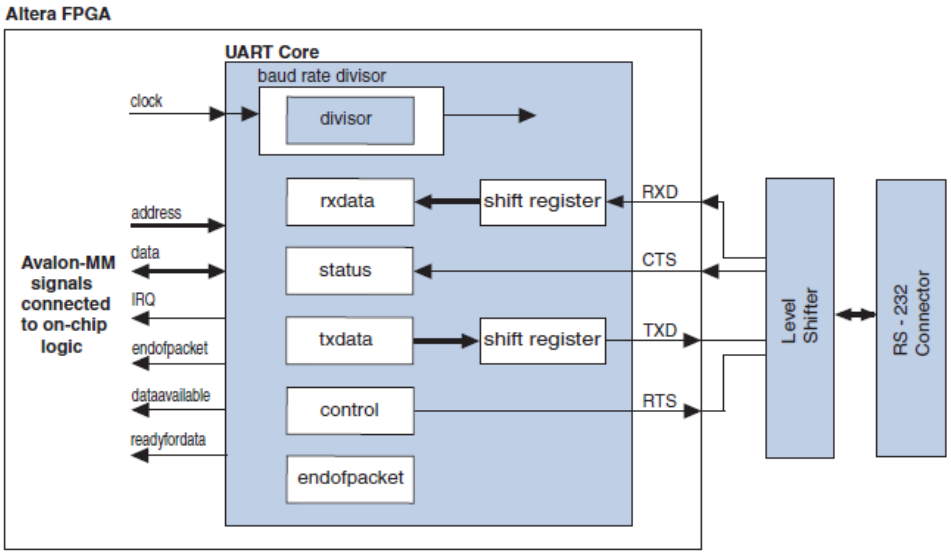


*Figure 4.17: Example connection using the UART core [278].*

- JTAG (Join Test Action Group) UART: this controller [278 – 279] provides a communication between the SoPC system built in the FPGA and a host PC, through the Avalon interface, using the JTAG circuitry inside the Altera FPGA, operating 8-bit data at a time. This Avalon interface consists of one 32-bit data register, and another 32-bit control register, accessed by an Avalon slave port. The connection is done with an USB

Blaster cable from the host PC to the FPGA, through several drivers provided in the HAL library [278 – 280]. Also, Altera provides software for managing the connection. Inside the FPGA, there are several JTAG nodes which are multiplexed through the single JTAG connection.

In Figure 4.18, we show an example system with one JTAG UART and the Nios II processor, which are connected using the USB Blaster cable to the host PC. Each host application has an independent connection.

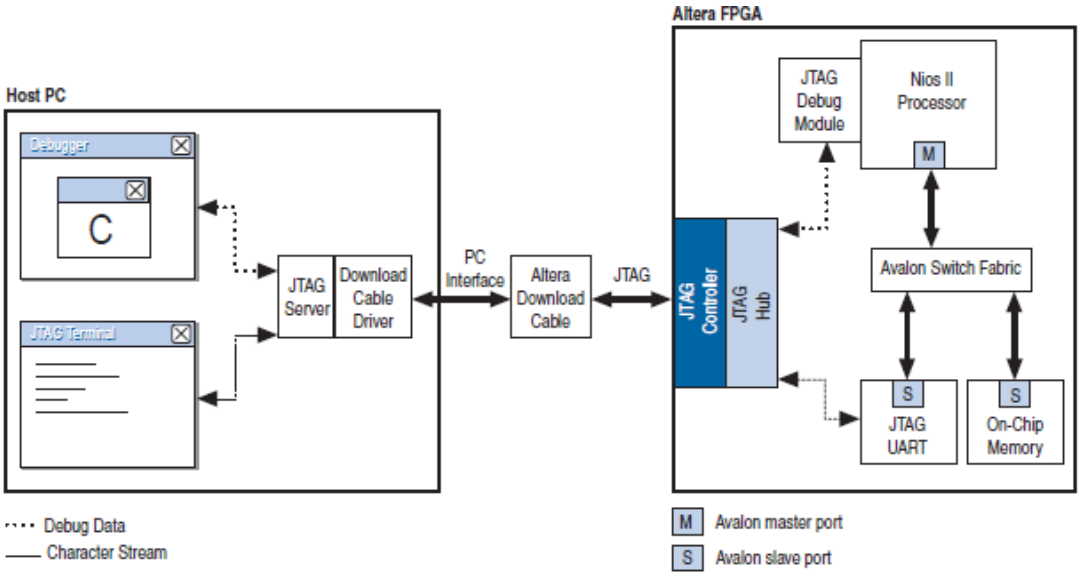


*Figure 4.18: Example connection using the JTAG UART core [278].*

- SPI (Serial Peripheral Interface) controller: this controller [278 – 280] provides a serial communication, through an Avalon interface, between the designed embedded system in the Altera FPGA, and a big variety of off-chip devices, like sensors, memories, or control devices among others. This controller can implement a master protocol, controlling up to 16 independent slaves, dividing the Avalon clock for generating the SCLK (Serial Clock) output signal, or a slave protocol synchronized to the SCLK (Serial Clock) input signal, both protocols with a flexible received/transmitted register width between 1 and 32 bits. The communication is achieved using a control line, two data lines, and a synchronized logic to the input clock.

In Figure 4.19, we present the SPI core configured as slave (left) and as master (right).

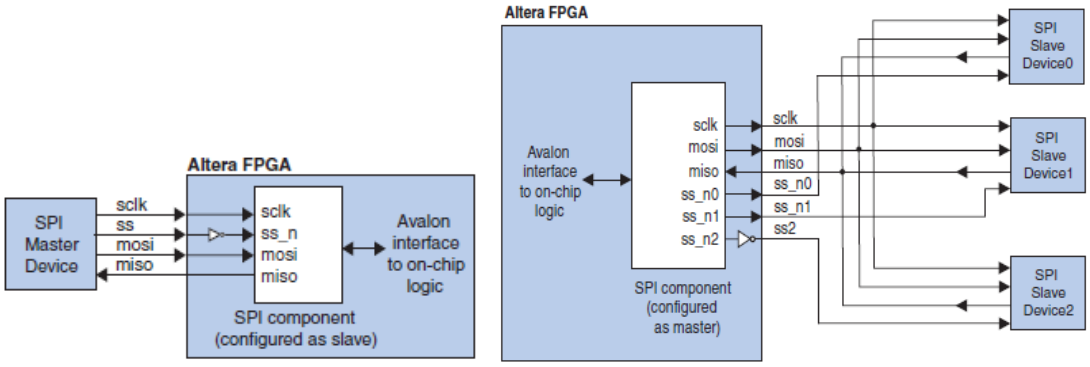


*Figure 4.19: SPI core configured as slave (left) and as master (right) [278].*

- PIO (Parallel Input/Output) controller: through parallel communication, this core [278 – 280] aim is to provide up to 32 connections for general purpose I/O ports, used to connect to on-chip user logic, off-chip devices, or external devices (LEDs, switches, or display devices between others). In fact, this controller can be configured for input, output, or both. As a connection, this controller uses a memory mapped interface between the ports and an Avalon slave port. The data management is done through IRQs (Interrupt Request) based on the input signals. The host will handle this data reading and writing the register mapped Avalon interface. In Figure 4.20, we show several connections through PIO cores.

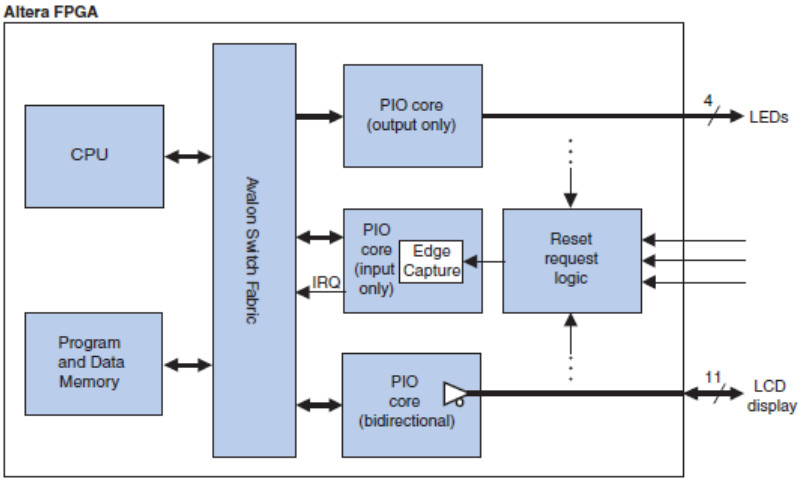


*Figure 4.20: Example using several PIO cores [278].*

- Timer: this core [278 – 280], defined as a 32-bit interval timer, can be connected through the Avalon interface to the Nios II processor system. Its controller provides controls to start, stop, or reset, the timer, two count modes, optional watchdog timer, or

maskable IRQ between other features. Also in this controller, Altera provides its drivers with the HAL system library. As an advantage of this peripheral, all the register are 16-bits wide, so it is compatible with 16-bit and 32-bit processors. Other advantage is that it can be configured in many ways, and depending on the selected configuration, the timer can be started and stopped, or only stopped. The timer behavior can be resumed in the following steps:

- Timer control registers are written:
  - Start/stop the timer.
  - Enable/disable IRQ.
  - Selected timer mode (continuous/once count down mode).
- A processor reads the status register.
- A processor can set the timer period.
- A processor can read the current counter value.
- When the counter reaches zero:
  - Generate IRQ whether enabled.
  - Optional watchdog resets the system.
  - Optional periodic pulse generator is asserted for one clock cycle.

In Figure 4.21, we show the timer core block diagram.

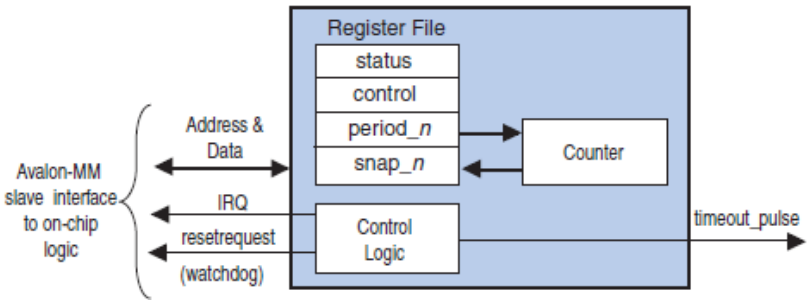


*Figure 4.21: Interval Timer core [278].*

### *4.5.3. Memory types and operating systems for SoPC.*

In this subsection, we are going to focus firstly on the different memory types [2.82] which we could use in the design of a SoPC system based on the Nios II processor. Secondly, we will focus on the different operating systems [289] which can be embedded in a SoPC system based on the Nios II processor.

Focusing on the first topic, the available memory types are the following four:

- On-chip memory: it is connected to the circuit board without using any external connection, since this kind of memory is embedded inside the FPGA. Due to this fact, this memory type is the fastest that can be used in an FPGA based embedded system, and serves the lowest possible latency. On-chip memory [282] has a high number of good characteristics, among which transactions pipelining or no additional circuit board wiring required are included. Some kinds of on-chip memories are characterized through dual port mode accessing with different port for reading and writing, which allows reading over one port while writing is done over the other port. Additionally, the cost of this memory is reduced because of its development inside the FPGA based embedded system. As drawbacks, its volatility makes it lose its contents when power is disconnected; and its limited capacity, because designed memory capacity depends only on the specific FPGA device. According to its advantages and disadvantages, on-chip memories are mainly used to store boot code or LUT (Look Up Tables).

- External RAM: external SRAM (Static Random Access Memory) [282] is implemented outside of the FPGA. The throughput of these memories remains high and the storage capacity is bigger than the previous family of memory commented. SRAMs usually have high performance and quite low latency. But its latency remains higher than that of On-chip memory because it connects to the FPGA trough a shared and simple bidirectional bus. It is feasible to share external SRAM buses between several memories, even when these memories belong to different families, such as flash or SDRAM. SRAM devices involve a higher cost per MByte and a larger board space than other memory types with high capacity like SDRAM. And of course, a larger board space than On-chip memory, which almost consume none.

- Flash memory: it is a nonvolatile memory type used frequently in machine vision embedded systems. It is mainly external to the FPGA, since FPGAs do not contain it. Because flash memory [282] retains the data after power off, it is used to hold microprocessor boot code as well as any data which needs to be preserved in the case of a power failure. Flash memory is not updated through a simple writing command, being needed a specific sequence of several read and write transactions. Additionally, before flash memory can be written, it must be erased. Entire sections of flash memory must be erased as a unit because individual words cannot be erased. This explains why the major disadvantage of flash is its writing speed. The actual write time can go up to microseconds due to the writing process requesting specific commands and bus transactions. Depending on clock speed, the actual write time can be in the hundreds of clock cycles. Because of the sector erase restriction, the write speed becomes poor and flash memory is mostly used only for storing data which will be retained after power off. These requirements sometimes make flash devices difficult to use. Actual image processing embedded systems use flash memory to keep also large data blocks plus the typical initial boot program. Again, embedded systems use flash memory as a hard drive, obtaining a platform with low power and high reliability. Flash memory has become popular since it is low cost, erasable, and durable. It is possible to connect flash buses with other flash devices, or even with external memories of other types, such as external SRAM or SDRAM.

- SDRAM (Synchronous Dynamic Random Access Memory): it is other type of volatile memory, similar to SRAM, but it must be refreshed periodically to keep its data. The devices working with SDRAM [282] are usually low-cost and with high capacity. Nevertheless, one specific hardware controller is needed to operate with it, since SDRAM organizes the memory space in columns, rows and banks, occupying the controller a major part of the interface. The complexity of the SDRAM interface requires always using the previous mentioned SDRAM controller, which drives the timing, address multiplexing, and cycle refreshing. Thus, SDRAM holds its large capacity keeping at the same time low cost. Additionally, the power consumption is lower comparing with its equivalent SRAM. It is feasible to share SDRAM buses for connecting many SDRAM devices, or even external memories of other families as flash or SRAM. Since SDRAM has an important amount of access latency, most SDRAM

controllers are the result of hard efforts to minimize this latency. Despite of this, SDRAM latency is always greater than the one of regular external SRAM or FPGA On-chip memory. However, the pipelining of consecutive accesses increases the SDRAM global performance after its high initial access. Some types of SDRAM can support higher clock rates than SRAM, improving its performance.

Now, we are going to focus on the second issue mentioned before: operating systems for a SoPC system. When developing a complex SoPC system it is needed the support to provide I/O synchronization, memory management, or multitask scheduling among other features. Due to the high complexity of its development, the designers of SoPC systems normally would use a third party developed instead of a personalized one. These operating systems [289] for embedded designs are usually simpler than their desktop counterparts, and usually provide tools for C/C++ software development. They also pretend to be RTOSs (Real Time Operating System). Although we are going to deal with only three of the operating systems supported by the Nios II, in Table 4.3 we show many more operating systems for embedded system designs:

| OS | RTOS | OS Type | Company Name | Nios II IDE Plug-in |
|---|---|---|---|---|
| eCos | Yes | Open Source | eCosCentric | - |
| Euros RTOS | Yes | Commercial | Euros | - |
| Erika Enterprise | Yes | Commercial | Evidence | Yes |
| ThreadX | Yes | Commercial | Express Logic | Yes |
| Nucleus Plus | Yes | Commercial | Mentor Graphics | - |
| MicroC/OS-II[16] | Yes | Commercial | Micrium | Yes |
| embOS | Yes | Commercial | Segger | - |
| osCAN[17] | Yes | Commercial | Vector Informatik | - |
| µClinux | - | Open Source | Microtronix | Yes |
| µClinux | - | Open Source | Community Supported (based on Microtronix port) | - |

*Table 4.3: Operating systems for embedded systems [283].*

The three main operating systems supported by the Nios II processor systems are described below:

- eCos (Embedded Configurable Operating System): it [289] is an open source customizable RTOS with multithread support but not multiprocess support. Due to the

fact that it is programmed under C, it is compatible with µITRON and POSIX. It was designed for small memory (hundreds of KB) systems with real-time requirements. Regarding its history, the first eCos was firstly bought by Cygnus solutions and later by RedHat. Because of this, eCos was transferred to the FSF (Free Software Fundation), and the RedHat development team continues developing eCosCentric.

- µClinux: it [289] is an open source operating system derived from Linux including libraries and tool chains, and destined to microprocessors like the Nios II processor. It supports different architectures making it useful for many embedded systems. Moreover, its development community keeps on developing patches and supporting tools.

- µC/OS-II: it [289] is a highly customizable RTOS with multitask support (up to 255 tasks) and destined to microprocessors and microcontrollers. It is programmed under C and used in a wide range of embedded devices. Its license is managed by Micrium which offers one free option for students. It has been ported as MicroC/OS-II to Altera Nios II processor systems. Because of this, programs developed under MicroC/OS-II constraints are easy portable to other hardware systems based on the Nios II architecture. In Figure 4.22 is shown how MicroC/OS-II is on the HAL system library top, using all the HAL services and APIs (Application Programming Interface).

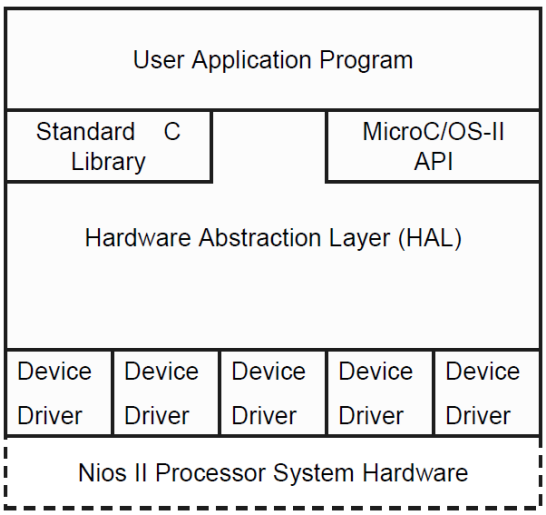


*Figure 4.22: MicroC/OS-II architecture [289].*

## 4.6. Future trends.

We are going to tackle the main embedded processors from the two main leading companies in the SoPC world on FPGA technology.

On the one hand, we have the already known Nios II embedded processor from Altera Company. In its last version [283] it has been improved with several features commented below:

- GCC upgrade to version number 4.7.3: using this version it is achieved a smaller size code.

- Floating point custom instruction support: this new custom instruction type is added to the previous existing types presented in the Qsys Tool.

- ECC (Error Correction Code) support on the RAM: it is only available inside the processor core and the instruction cache on the Nios II/f processor when configured without data cache.

- RTL (Register Transfer Language) trace simulation support: it allows the developer to record and measure time in the instruction execution process of the Nios II processor during the RTL simulation in ModelSim. It provides changes, control, and supervision of the data address, instruction execution, interrupts, and control registers in the hardware simulation.

- Impulse C compiler: this compiler provides software to hardware technology for C language to use with the Nios II embedded processor. Using this new tool, the user can develop Avalon-MM (Avalon Memory Mapped) hardware components and use then when designing and embedded system.

On the other hand, we have the also known MicroBlaze [284] embedded processor from the Xilinx Company. In its last version it has been improved with several features commented below:

- Sleep instruction support: this improves its performance when looking for low power systems.

- Re-locatable addresses for base vector: this provides a higher flexibility in memory sharing.

- I/O improvements: in this section we can find GPI (General Purpose Input) interrupts and programmable baud rate in UART.

- Multiple LMBs (Local Memory Bus) support.

- Lower latency interrupts support (up to 10X): this is achieved when the interrupt vector controls directly an individual interruption.

- Byte-swap instruction support: this is really appreciated when improving the Ethernet performance.

- Addable AXI System Cache: this makes a selected memory behaves as level two cache, improving systems performance in a big rate.

Because this last embedded processor [284] has not been deeply examined during this work, we present below in Figure 4.23 the different performances achieved by the Xilinx MicroBlaze embedded processor.

| MicroBlaze Processor v8.40.b Performance Levels (v14.4 XPS) | | | |
|---|---|---|---|
| **Device Family** | **Performance Optimized MicroBlaze with branch optimizations (5-stage pipeline) 1.38 DMIPs/MHz** | **Performance Optimized MicroBlaze (5-stage pipeline) 1.30 DMIPs/MHz** | **Area Optimized MicroBlaze (3-state pipeline) 1.03 DMIPS/MHz** |
| Zynq-7000 SoC (-3) | 228DMIPs | 259DMIPs | 196DMIPs |
| Virtex-7 FPGA (-3) | 293DMIPs | 393DMIPs | 264DMIPs |
| Kintex-7 FPGA (-3) | 317DMIPs | 408DMIPs | 264DMIPs |
| Virtex-6 FPGA (-3) | 306DMIPs | 384DMIPs | 246DMIPs |
| Spartan-6 FPGA (-4) | 166DMIPs | 209DMIPs | 152DMIPs |

*Figure 4.23: Xilinx MicroBlaze achieved performances [284].*

# Chapter V

## Accelerating through C2H

This chapter tackles the improvement of our system thanks to the Altera C2H compiler, which accelerates the motion estimation code. We start this chapter with an introduction of this paradigm; additionally a profiling of the algorithms applied is done. Finally, are shown the achieved results using different metrics, through a detailed explanation of the different system qualities, and several comparisons with other current works. The system developed here is capable to deal efficiently with 72.5 KPPS, equivalent to a SoC which processes 50×50 @ 29.5 fps.

## 5.1. Introduction.

In this chapter, the acceleration of motion estimation techniques using the Altera C2H (C to Hardware) paradigm is shown. The C2H compiler [295] is a feature provided by the Altera design suite [304], whose task is to build hardware accelerators from the ANSI C [302] source code provided, improving the performance of the programs running on the Nios II processor. This engine improves the throughput and hardware design, as it will be shown in this work, due to the fast and simple substitution of the chosen software functions with independent hardware accelerators, which are placed into the design downloaded to the FPGA (Field Programmable Gate Array)[1] [307]. Additionally, C2H [295] will produce an exhaustive report including used resources, throughput, and hardware structure.

C2H is included in the Altera design suite, inside the Nios II IDE (Integrated Development Environment) [303], and it works together with the different tools inside the Altera framework.

Each function chosen to be speeded up will be translated into one hardware accelerator if this function complies with all the C2H requirements. These hardware accelerators will be added automatically, using the SOPC Builder functionality, to reach the Nios II processor design, through the Avalon switch fabric [298 – 299], like other peripherals in the system. As evidence of the automatic design of these hardware accelerators, they are provided with the following features:

- Direct memory access: same way of access as the Nios II processor.
- Pipelined loops: possible code is executed in parallel.
- Pipelined memory access: reducing the latency in memory access.

As a drawback, the C2H compiler only supports ANSI C source code and even some ANSI C parts are not included, but these parts are detailed in the Altera C2H specifications [295]. Despite of this, C2H can add to the design a hardware accelerator, which replaces the pure software function. To achieve an optimum improvement, the C2H compiler translation is explained in the next sections. C2H follows a design flow [295] which is described below:

1) The GCC preprocessor [305] evaluates preprocessor directives: these directives are those source code lines after a hash sign (#), and they are directives for the preprocessor and not program statements. Preprocessing is the first step when transforming source code into an executable file [306]. During this step, the preprocessor examines the source code looking for preprocessor directives before compilation, which is the next step. These preprocessor directives extend only through a single line of code without expecting semicolon (;) at the end. Indeed, when a newline character is found, the preprocessor directive ends. The only way to use more than one line for a preprocessor directive is by preceding the newline character with a backslash (\) symbol.

2) Code parsing: this step consists on parsing the source code, after the preprocessor finishes evaluating the preprocessor directives, in a set of objects, like a standard compiler.

3) Dependencies graph [295]: using the set of objects resulting from the previous step, the C2H tool first creates a dependencies graph, for transforming this graph into a state machine, where each assignment results into a state. This state machine manages the sequence of operations defined by the source code function which is being accelerated.

4) Optimizations: firstly, the C2H tool converts directly each element of the source code into an equivalent hardware structure using simple translation rules. But later, during this step, the C2H compiler performs several optimizations for reducing used logic elements through resource sharing, achieving a better result than a one to one mapping.

5) Best sequence: in this step, C2H searches for the best sequence to perform any operation in the accelerated function.

6) Generation of the hardware accelerator: here, the C2H tool generates a synthesizable HDL (Hardware Description Language) [308] object file which defines the generated hardware accelerator.

7) Generate a software wrapper [295]: in this step, C2H creates a C wrapper function to control and interact with the generated hardware accelerator, reading and writing the register interface. Thus, the details of the interaction between the hardware accelerator and the Nios II processor remain isolated and hidden. Indeed, from the perspective of the calling function, the result of calling the software wrapper is

functionally the same as calling the original C function, as the wrapper function is a C file that replaces the original C function at software link time, operating as follows:

1. Accelerator parameters set up: this corresponds to passing and setting variables in the original non accelerated (sequential) function.
2. Processor's data cache flushing: this step is optional, and it is only performed when the hardware accelerator accesses the same memory that the processor does.
3. Start of the accelerator: runs the hardware accelerator which, depending on its functionality, can return a value, terminate, or run persistently.
4. Register polling: they are polled to determine when the hardware accelerator task is finished.
5. Return of the result: in this step the result value is read if the hardware accelerator returns it, for later passing it to the calling function.

In Figure 5.1, it is shown how the C2H hardware accelerators are included into a Nios II processor based system, although in this case there is only one hardware accelerator included. Altera's Avalon Switch Fabric [298 – 299] works as a bus standard, which interconnects the different components in a Nios II processor based system, allowing them to transfer data. The Avalon interface used in Figure 5.1 to interconnect the hardware accelerator and the different system components is the Avalon-MM (Memory Mapped) interface [298]. This interface follows the master-slave paradigm, where the masters can initiate a transaction, and the slaves respond to the masters' requests and are able to generate return data. The presented MUX (Multiplexor) allows selecting the data from the desired slave, while the bus arbitrators decide which master gains control on each case.

As a side effect, the hardware accelerators consume hardware resources from the FPGA (Field Programmable Gate Array), such as logic elements or on-chip memory, which will not be available for other components in the design.

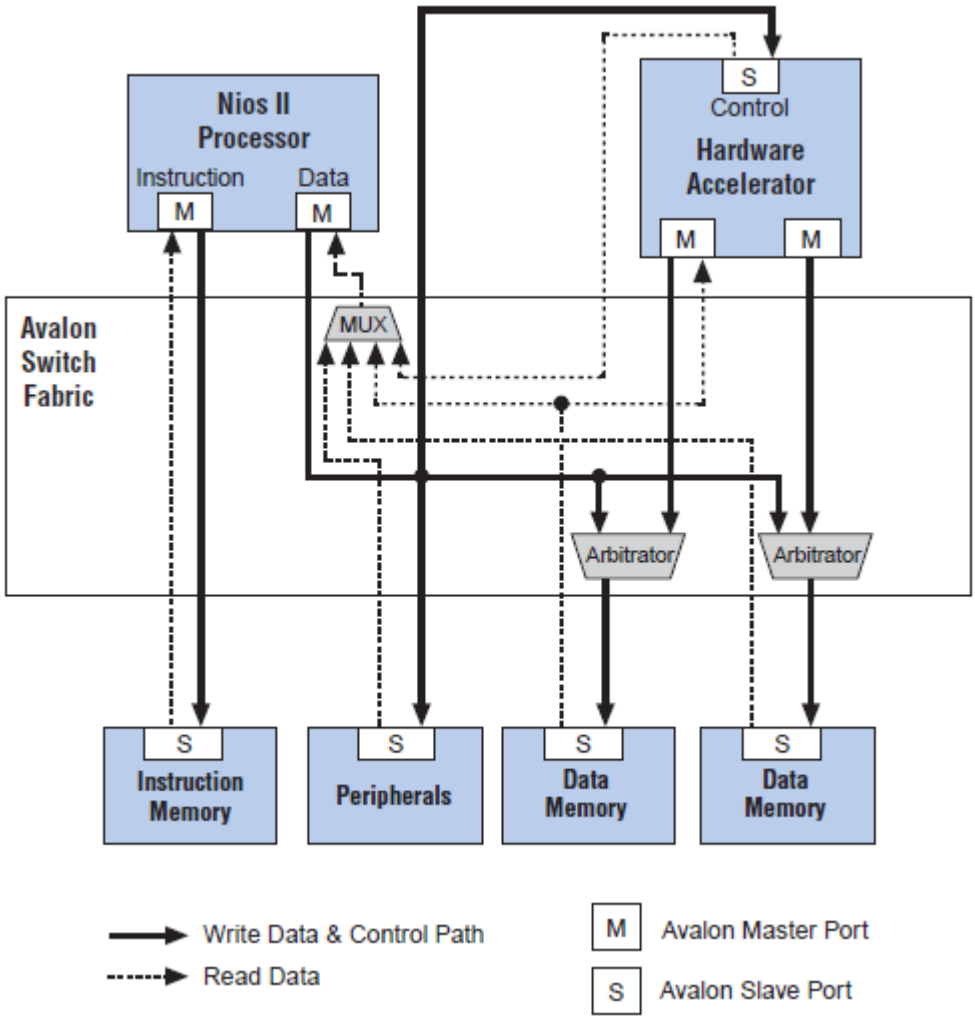


*Figure 5.1: Example system using one hardware accelerator [295].*

After the introduction of the Altera C2H engine, the next section will describe in a detailed way how this tool works, and what are the requirements we must meet for a better hardware accelerator, either looking for reducing used hardware resources or looking for an improved execution speed.

## 5.2. Topology and architectural description of the accelerator.

In this section, the internal working of the C2H module is described, and how it achieves the best performance for the acceleration of individual software functions written in ANSI C [302] source code.

There are three different improvement criteria when accelerating a function using the C2H environment: the first one is the number of hardware resources used, the second

one is the performance of the accelerated function, and the last one is the impact of the hardware accelerator in the whole system.

Although a usual C compiler translates the source code into instructions, which access the hardware resources in a shared way, the C2H tool takes as input the C source code assuming a sequential computing model. It translates this C code into one or more state machines which access the hardware resources in an exclusive way, pipelining the generated code as much as possible for improving its performance. For example, the C2H will translate two different sums which are added in a final sum, into two adders which would execute the two first sums in parallel, and a final adder for the final sum, instead of translating the three sums into three different instructions executed sequentially in the same adder.

Now we are going to present how the C2H module translates the different C structures [296]:

- Memory accesses: the C2H tool creates several registers made of logical elements to store the different scalar variables in the function. This is very fast and not too expensive in terms of logical elements. It also creates some logic to address calculation, and an Avalon-MM (Memory Mapped) [298] master port to access it, when using arrays or pointer dereferences. This is expensive, and its execution speed depends on the memory type the master port is connected to.

- Arithmetic/Logical Operations: C2H works different to a usual C compiler. For example, shifts operations are made through wires, and due to this, these operations do not have any hardware cost and are constant in time complexity. Moreover, multiplies and divisions by a power of two, and bitwise ORs and ANDs by a constant, are made with wires too. Due to this fact, they are constant and free too. In other cases such as divisions, it requires some logic or even specific circuitry.

- Statements: a C compiler translates expressions with many operators into instructions, but the C2H tool translates them to a path. Due to this fact, we have to be very careful coding our program, because very large expressions could produce large paths inside the hardware accelerators, and these paths will be very expensive and degrade the maximum frequency of the whole design.

- Control flow: “if” statements are translated into control logic controlled by the questioned expression. Loops, including “do”, “while”, and “for”, are pipelined to achieve one iteration in one cycle. In loops, the evaluation expression can cost another cycle if it has data dependency with any statement within the iteration. Otherwise, switch statements are translated to “if” statements.

- Subfunction calls: in the C2H tool the subfunction calls are usually translated into new accelerators, which is very expensive and performance degrading. But, if the subfunction has an immovable execution time, their calls are pipelined inside the accelerated main function.

- Resource sharing: due to the fact that the C2H tool translates our C code into different hardware resources, it is in the developer’s hands to create subfunctions with the operations that he/she wants to share their hardware resources.

- Dependencies of data: data dependencies are managed in the C2H tool replacing concurrent operations by sequential ones.

- Memory architecture: the designer has to be very careful when developing the function source code because dereferences create Avalon-MM [298] master ports, and these could be all connected to the same Avalon-MM slave port. In these cases, the performance is degraded. The memory interface acts as a bottleneck because an Avalon-MM slave port can arrange just one write or read operation at a given time. Another issue to take into account is that when a hardware accelerator accesses memory, it does it through its Avalon-MM master port, bypassing the Nios II data cache.

As an advantage, we can choose on each hardware accelerator whether the Nios II processor flushes all the data cache when calling it. A drawback could be that the DRAM controller has to cope with non sequential accesses: many Avalon-MM master ports could access it at the same time, but the DRAM controller only keeps one memory bank open at any given time. In these cases the arbiter uses the round robin technique by default.

In Figure 5.2, we show how the C2H module works and its position inside the design process. The left half of the flowchart shows the standard C compilation of “main.c”

and "accelerator.c", as it occurs without acceleration. It also shows the generation and selective linking of the accelerator driver into the executable file. The right half of the flowchart shows the hardware compilation process invoked when a function in "accelerator.c" is accelerated with the C2H compiler.

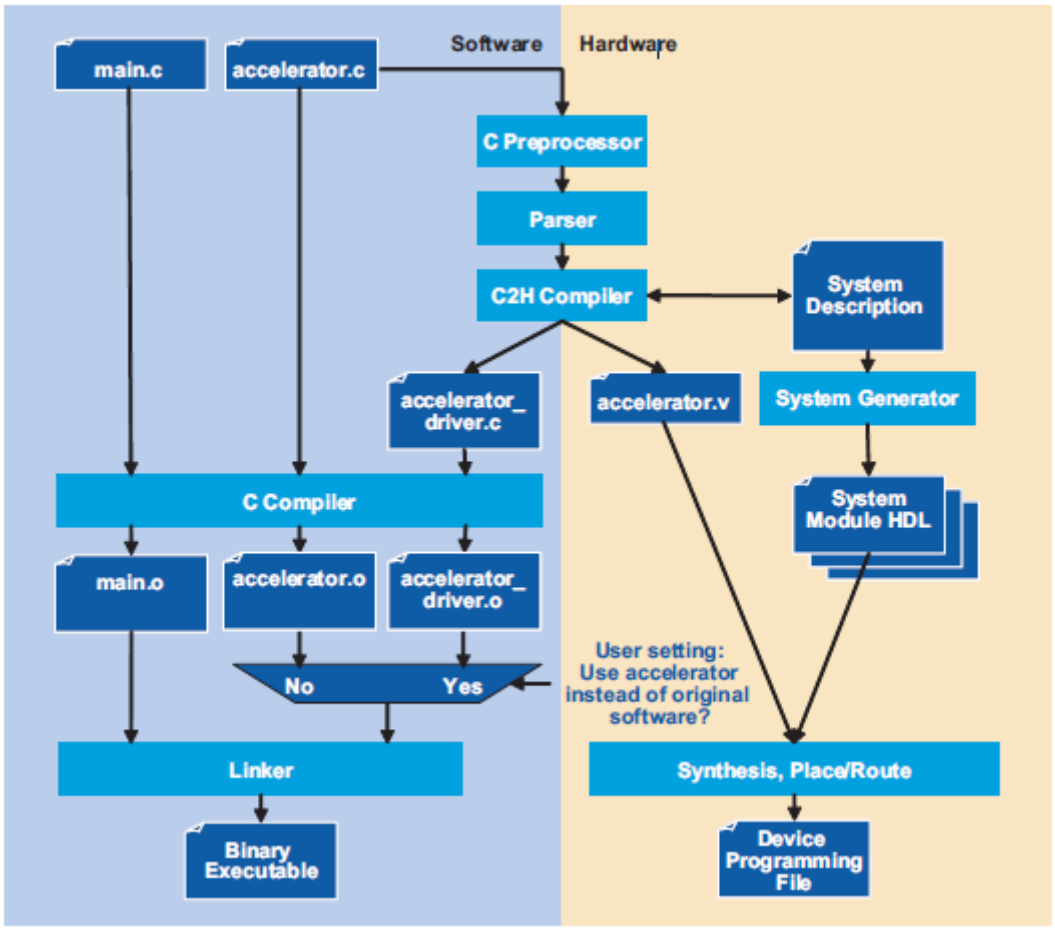


*Figure 5.2: C2H integration designing a system [300].*

And now, we show in Figure 5.3 how the hardware accelerator generated by the C2H tool is integrated in the designed system thanks to the Avalon Switch Fabric [298 – 299]. As commented before, Altera's Avalon Switch Fabric works as a bus standard interconnecting the different components in a Nios II processor based system, even the generated hardware accelerator. Connected components through the Avalon-MM (Memory Mapped) interface are able to transfer data using the master-slave paradigm, where the master can initiate a transaction, and the slave responds to the master's requests and is able to generate return data. Arbiters are present where several masters

have to be connected to the same slave component, deciding which master gains control on each case.

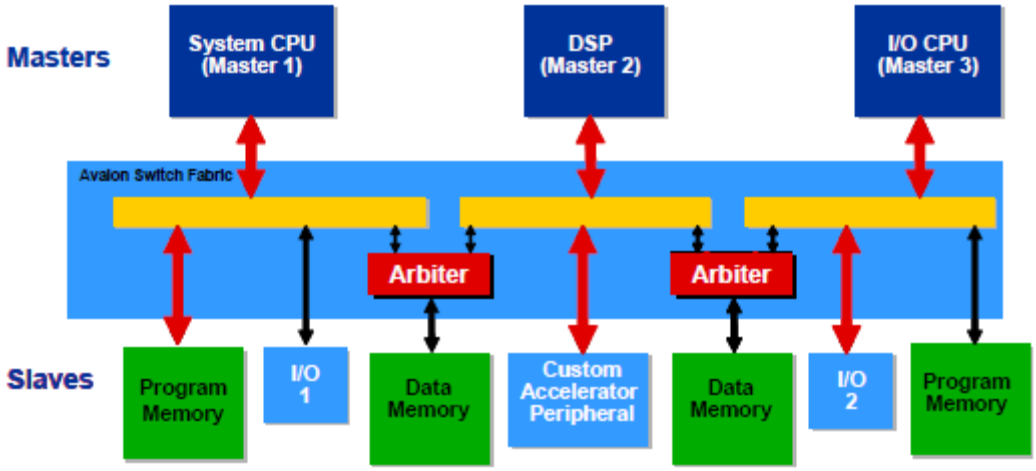


*Figure 5.3: Integrated hardware accelerator with Avalon Switch Fabric [300].*

### 5.2.1. C2H integration with Avalon bus and Master Slave paradigm.

In this subsection, we are going to explain the interconnection of the hardware accelerators generated by the C2H and the system based on the Nios II through the Avalon system.

Avalon interfaces [298], which are an open standard, have the aim to provide simple interconnections between the different components forming the system in an Altera FPGA. Between these components we can find memories, registers, and off-chip devices among others. There is not a one to one correspondence between components and interfaces; indeed it is a many to many relationship. Each Avalon interface has different signals, some optional, which provides a high flexibility in the component design process, including bursting (multiple transferences as one) or packet support. The interfaces performance is not guaranteed due to the dependence on its implementation and its design. A design recommendation provided by Altera is that these interfaces should not be edge sensitive apart from reset or clock signals, due to the signal stabilization process. Moreover every Avalon interface is synchronous.

There are seven different types of interfaces which are going to be briefly described here:

- Avalon-ST (Streaming): this interface [298] is specialized in managing data flow in only one direction, which include for example packets or DSP (Digital Signal Processed) data.

- Avalon-MM (Memory Mapped): this address based interface [298] is typical of connections based on the master-slave paradigm.

- Avalon Conduit: this interface [298] is mainly used to group a collection of signals, which do not fit into any other Avalon interface type.

- Avalon-TC (Tri-State Conduit): this interface [298] provides connection to off-chip peripherals, even when several peripherals use the same signal, sharing pins.

- Avalon-II (Interrupt Interface): this interface [298] provides events from one component to another through a signal.

- Avalon-CI (Clock Interface): this interface [298] provides clock support.

- Avalon-RI (Reset Interface): this interface [298] provides reset support.

Once interfaces have been described, we are going to focus on the Avalon MM interface due to the fact that it is the one used for connecting C2H generated hardware accelerators.

Avalon can define interfaces for master-slave connections through mapped memory supporting fixed cycle, pipelined, or burst read/write. The mentioned interface connects components such as memories, microprocessors, timers, or UARTs (Universal Asynchronous Receiver Transmitter). When configuring these interfaces, it is only needed to include the required signals for the component logic. These signals keep a one to one relationship with the Avalon MM signal roles in an Avalon MM port. As told before, these interfaces are synchronous, thanks to synchronization through a linked clock interface. In the Avalon MM interfaces, transferences are begun by the master port, interacting with the slave port thanks to control and data signals through the interconnect fabric. These transfers can take more than one clock cycle and be sized from 1 to 128 bytes.

In Figure 5.4, we show an example of connecting a custom logic block to the interconnect fabric using the Avalon-MM interface.

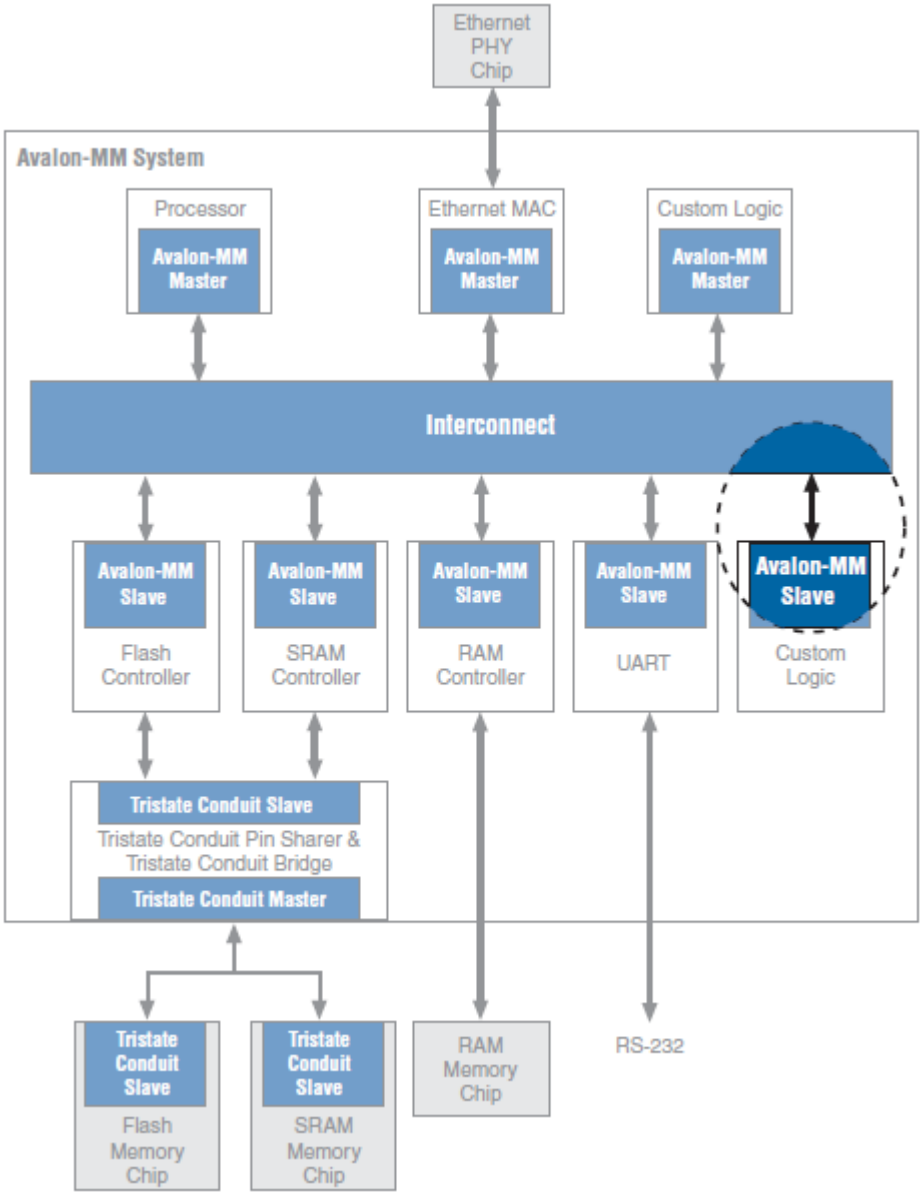


*Figure 5.4: Example of using Avalon-MM interfaces [298].*

## 5.3. Proposed architectures.

In this section, several algorithms accelerated through the C2H tool are explained. Three accelerated techniques have been previously presented in Section 3.2.1. For a better understanding of these algorithms, we are going to present their data flows and also, data flows of the accelerated functions inside them. We have worked in the same

manner for every algorithm, executing them for each macroblock in the desired frame. These macroblocks, are used to look for the closest macroblock to each of them, searching in the reference frame, and using an error metric to measure the similarity between macroblocks. Following, we show the data flows used for each algorithm in every execution.

The FST algorithm data flow is shown in the Figure 5.5.

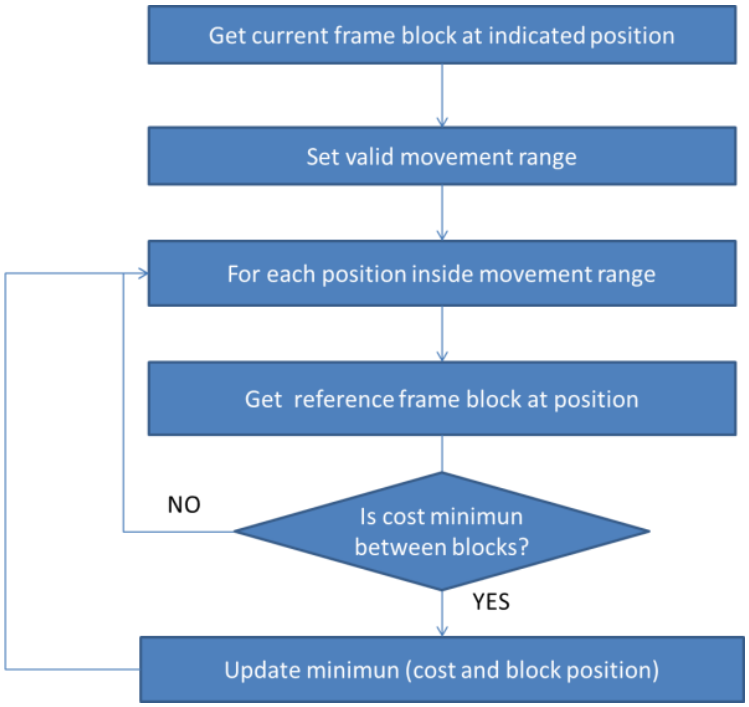


*Figure 5.5: FST data flow [297].*

The TSST algorithm data flow is presented below in Figure 5.6.

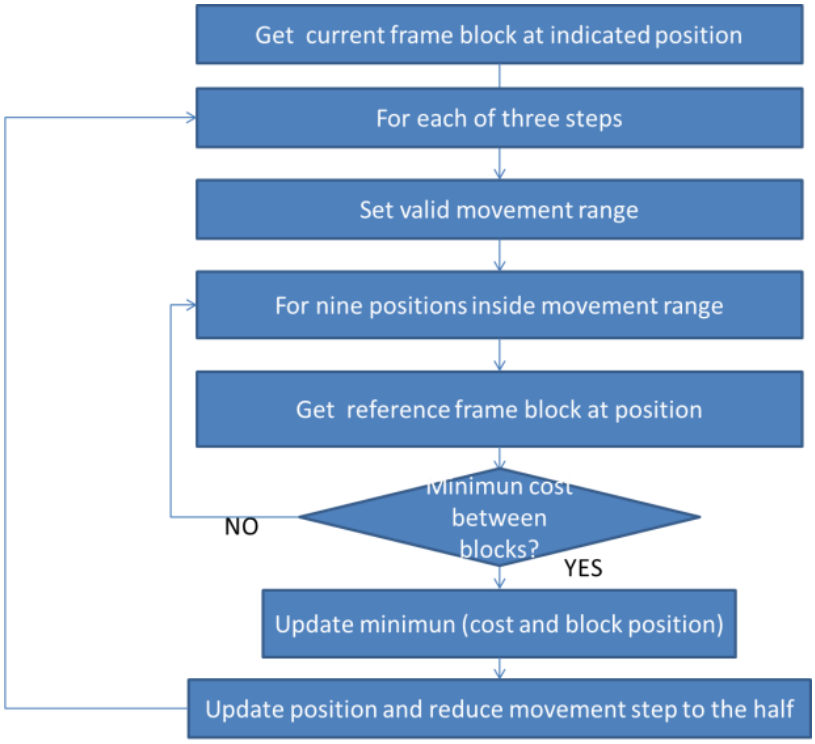


*Figure 5.6: TSST data flow [297].*

Now, the 2DLOG algorithm data flow is shown in Figure 5.7.

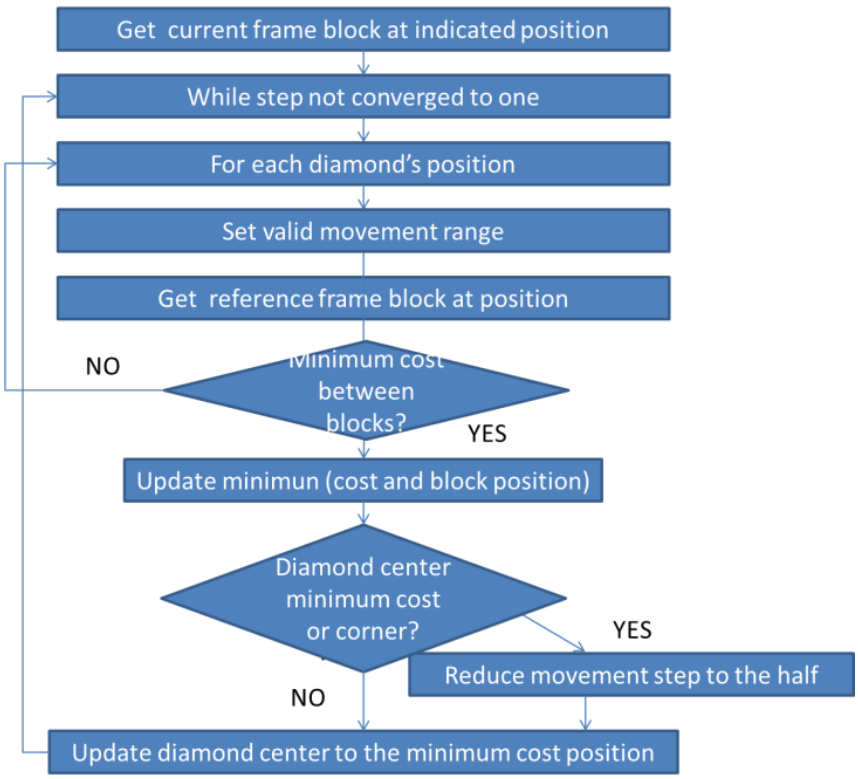


*Figure 5.7: 2DLOG data flow [297].*

Due to the fact that our work is focused on the topic of low-cost, we require a balance between used hardware resources and obtained throughput. For this reason, we have to divide the algorithms into several functions, with which we can achieve different acceleration qualities. These functions are described below, and their data flow is also presented for a better overall view of them:

- CopyBlock: this function aim is to copy a fixed number of bytes from one source address to a destination address. Its data flow is presented in Figure 5.8.

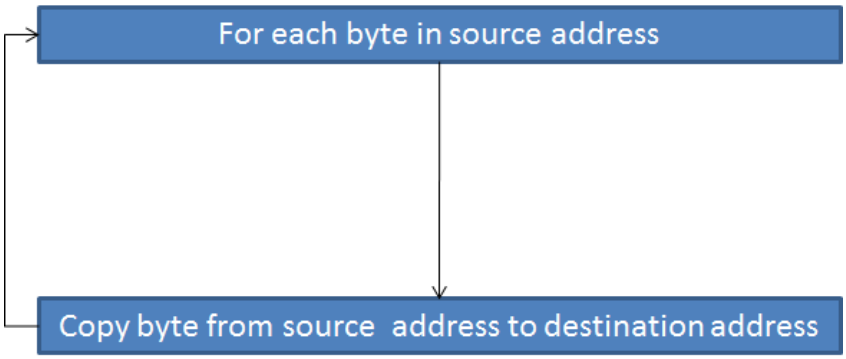


*Figure 5.8: CopyBlock data flow [297].*

- GetBlock: this function performs the copy of a macroblock from a source address to a destination address, though several calls to CopyBlock function. Its data flow is presented in Figure 5.9.

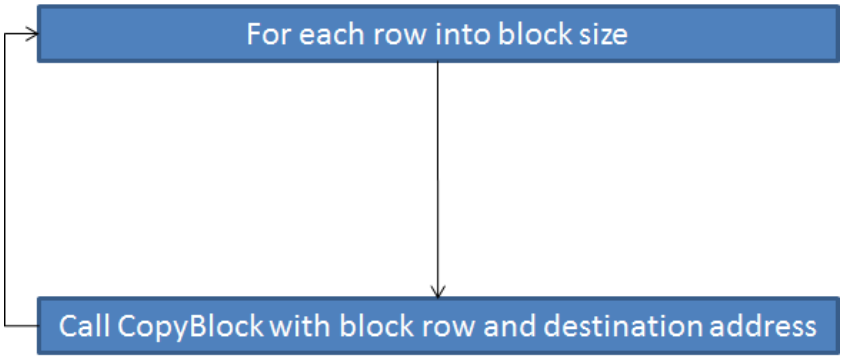


*Figure 5.9: GetBlock data flow [297].*

- GetCost: this function aim is to calculate the cost between two macroblocks for a given metric. Its data flow is presented in Figure 5.10.

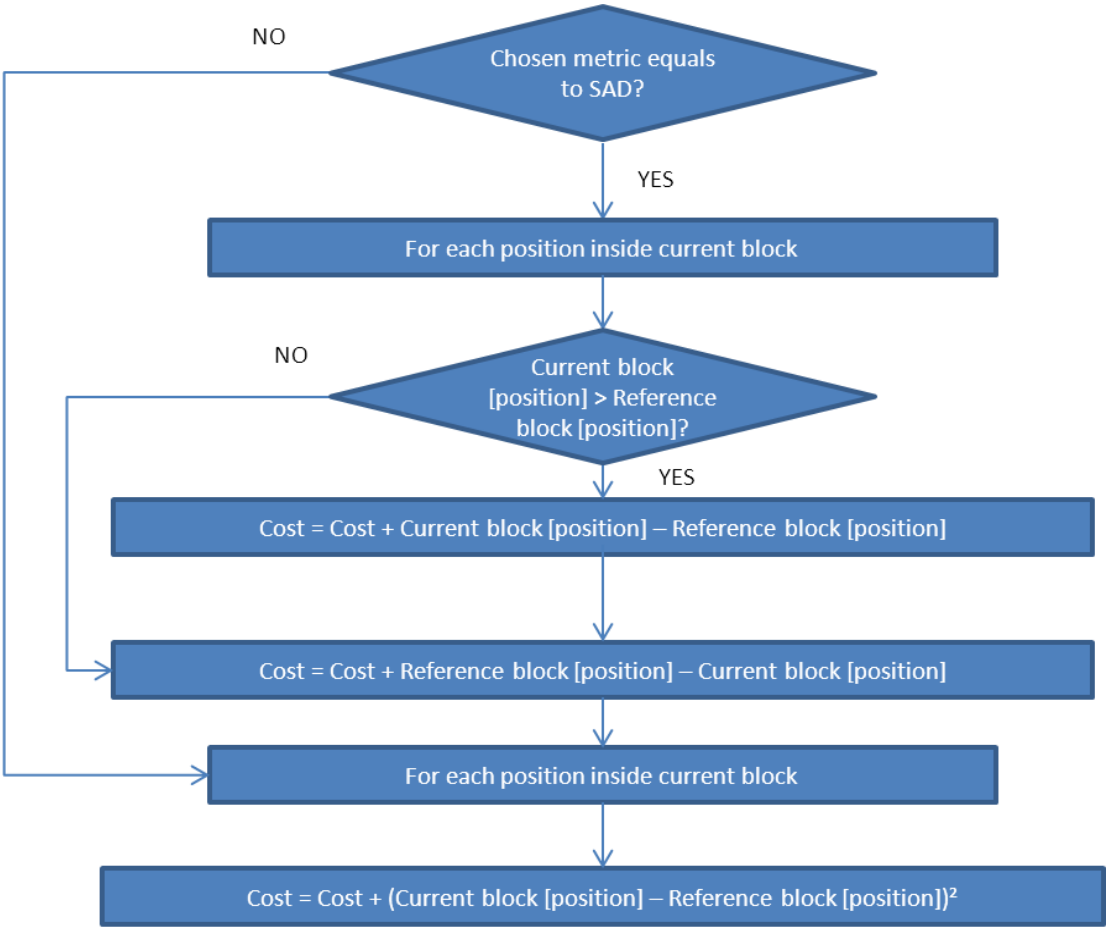


*Figure 5.10: GetCost data flow [297].*

### *5.3.1. Profiling.*

To distinguish the weakest point in our three presented algorithms (FST, 2DLOG, and TSST), we have performed a complete profiling of each one of these three techniques. This profiling has been done under the same well-known sequences ("Foreman" and "Carphone" presented in Section 3.3.1) for every presented macroblock size (16×16, 32×32, and 64×64) and additionally presented window size (8×8, 16×16, and 32×32).

In Tables 5.1 to Table 5.3 the mentioned profiling using the "Foreman" sequence is presented for the three motion estimation techniques (FST, 2DLOG and TSST), where it is shown the number of calls and the percentage (%) of time spent by these calls for the functions CB (CopyBlock), GB (GetBlock), GC (GetCost), and FN (FST, 2DLOG,

or TSST depending on the used algorithm). Additionally it is performed again the mentioned profiling using the "Carphone" sequence, showing it in Tables 5.4 to 5.6.

| Window size | Macroblock size 16 | | | | Macroblock size 32 | | | | Macroblock size 64 | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | **CB** | **GB** | **GC** | **FN** | **CB** | **GB** | **GC** | **FN** | **CB** | **GB** | **GC** | **FN** |
| 8 | 376416 (12.5) | 23526 (~0) | 23427 (87.5) | 99 (~0) | 148800 (16.67) | 4650 (~0) | 4620 (83.33) | 30 (~0) | 43480 (~0) | 685 (~0) | 676 (~100) | 9 (~0) |
| 16 | 1405024 (7.41) | 87814 (7.41) | 87715 (85.19) | 99 (~0) | 562560 (4.35) | 17580 (~0) | 17550 (95.65) | 30 (~0) | 163776 (8.33) | 2559 (~0) | 2550 (91.67) | 9 (~0) |
| 32 | 4844640 (8.57) | 302790 (0.48) | 302691 (90.95) | 99 (~0) | 2155392 (8.57) | 67356 (1.43) | 67326 (90.00) | 30 (~0) | 604096 (14.63) | 9439 (~0) | 9430 (85.37) | 9 (~0) |

*Table 5.1: FST profiling [320].*

| Window size | Macroblock size 16 | | | | Macroblock size 32 | | | | Macroblock size 64 | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | **CB** | **GB** | **GC** | **FN** | **CB** | **GB** | **GC** | **FN** | **CB** | **GB** | **GC** | **FN** |
| 8 | 26384 (~0) | 1649 (~0) | 1550 (~100) | 99 (~0) | 10784 (~0) | 337 (~0) | 307 (~100) | 30 (~0) | 3968 (~0) | 62 (~0) | 53 (~100) | 9 (~0) |
| 16 | 35616 (~0) | 2226 (~0) | 2127 (~100) | 99 (~0) | 14016 (~0) | 438 (~0) | 408 (~100) | 30 (~0) | 4992 (~0) | 78 (~0) | 69 (~100) | 9 (~0) |
| 32 | 43280 (~0) | 2705 (~0) | 2606 (~100) | 99 (~0) | 25952 (~0) | 811 (~0) | 781 (~100) | 30 (~0) | 8640 (~0) | 135 (~0) | 126 (~100) | 9 (~0) |

*Table 5.2: 2DLOG profiling [320].*

| Window size | Macroblock size 16 | | | | Macroblock size 32 | | | | Macroblock size 64 | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | **CB** | **GB** | **GC** | **FN** | **CB** | **GB** | **GC** | **FN** | **CB** | **GB** | **GC** | **FN** |
| 8 | 38976 (~0) | 2436 (~0) | 2337 (~100) | 99 (~0) | 19040 (~0) | 595 (~0) | 565 (~100) | 30 (~0) | 8640 (~0) | 135 (~0) | 126 (~100) | 9 (~0) |
| 16 | 38928 (~0) | 2433 (~0) | 2334 (~100) | 99 (~0) | 19040 (~0) | 595 (~0) | 565 (~100) | 30 (~0) | 8384 (~0) | 131 (~0) | 122 (~100) | 9 (~0) |
| 32 | 38832 (~0) | 2427 (~0) | 2328 (~100) | 99 (~0) | 20480 (~0) | 640 (~0) | 610 (~100) | 30 (~0) | 8640 (~0) | 135 (~0) | 126 (~100) | 9 (~0) |

*Table 5.3: TSST profiling [320].*

| Window size | Macroblock size 16 | | | | Macroblock size 32 | | | | Macroblock size 64 | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | **CB** | **GB** | **GC** | **FN** | **CB** | **GB** | **GC** | **FN** | **CB** | **GB** | **GC** | **FN** |
| 8 | 376416 (14.29) | 23526 (~0) | 23427 (85.71) | 99 (~0) | 148800 (6.67) | 4650 (~0) | 4620 (93.33) | 30 (~0) | 43480 (~0) | 685 (~0) | 676 (~100) | 9 (~0) |
| 16 | 1405024 (13.04) | 87814 (~0) | 87715 (86.96) | 99 (~0) | 562560 (11.76) | 17580 (~0) | 17550 (88.24) | 30 (~0) | 163776 (~0) | 2559 (~0) | 2550 (~100) | 9 (~0) |
| 32 | 4844640 (3.57) | 302790 (2.86) | 302691 (93.57) | 99 (~0) | 2155392 (9.32) | 67356 (~0) | 67326 (90.68) | 30 (~0) | 604096 (14.29) | 9439 (2.86) | 9430 (82.86) | 9 (~0) |

*Table 5.4: FST profiling [320].*

| Window size | Macroblock size 16 | | | | Macroblock size 32 | | | | Macroblock size 64 | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | CB | GB | GC | FN | CB | GB | GC | FN | CB | GB | GC | FN |
| 8 | 26464 (~0) | 1654 (~0) | 1555 (~100) | 99 (~0) | 10624 (~0) | 332 (~0) | 302 (~100) | 30 (~0) | 3648 (~0) | 57 (~0) | 48 (~100) | 9 (~0) |
| 16 | 34256 (~0) | 2141 (~0) | 2042 (~100) | 99 (~0) | 13696 (~0) | 428 (~0) | 398 (~100) | 30 (~0) | 4672 (~0) | 73 (~0) | 64 (~100) | 9 (~0) |
| 32 | 41088 (~0) | 2568 (~0) | 2469 (~100) | 99 (~0) | 25312 (~0) | 791 (~0) | 761 (~100) | 30 (~0) | 8256 (~0) | 129 (~0) | 120 (~100) | 9 (~0) |

*Table 5.5: 2DLOG profiling [320].*

| Window size | Macroblock size 16 | | | | Macroblock size 32 | | | | Macroblock size 64 | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | CB | GB | GC | FN | CB | GB | GC | FN | CB | GB | GC | FN |
| 8 | 39072 (~0) | 2442 (~0) | 2343 (~100) | 99 (~0) | 18944 (~0) | 592 (~0) | 562 (~100) | 30 (~0) | 8576 (~0) | 134 (~0) | 125 (~100) | 9 (~0) |
| 16 | 38928 (~0) | 2433 (~0) | 2334 (~100) | 99 (~0) | 18944 (~0) | 592 (~0) | 562 (~100) | 30 (~0) | 8576 (~0) | 134 (~0) | 125 (~100) | 9 (~0) |
| 32 | 38320 (~0) | 2395 (~0) | 2296 (~100) | 99 (~0) | 20384 (~0) | 637 (~0) | 607 (~100) | 30 (~0) | 8896 (~0) | 139 (~0) | 130 (~100) | 9 (~0) |

*Table 5.6: TSST profiling [320].*

By examining the profiling results, some conclusions can be reached about the most appropriate part of the source code to be replaced by a specific custom instruction. The GetCost function was the first in line; moreover, as shown in the previous tables, almost all the execution time was taken by it.

### *5.3.2. Acceleration qualities classification.*

For future use of this work, and for giving the developer the choice of the balance between hardware cost and performance, this work has split the acceleration of the presented algorithms into four different levels, which are described below:

- No [acceleration]: all source code is executed as software and no hardware accelerators are placed in the design.

- Low: only the CopyBlock function is executed through a hardware accelerator generated by the C2H tool. Remaining source code, including GetBlock and GetCost functions, is executed by software translated to Nios II processor instructions.

- Medium: the functions CopyBlock and GetBlock are executed through hardware accelerators generated by the C2H tool and placed in the design. Remaining source code is executed as software.

- High: all the presented functions are executed by their respective hardware accelerators generated by the C2H tool. These functions are CopyBlock, GetBlock, GetCost, and the block-matching algorithms (FST, 2DLOG, and TSST). Remaining source code, that is the program structure which calls the algorithms, is executed by software translated to Nios II processor instructions.

## 5.4. Results for the presented architectures.

In this section, we tackle different approaches to present the achieved results when using hardware accelerators. To offer a wider view about the possible results, these are going to be measured applying different block-matching techniques (FST, 2DLOG and TSST), different window search sizes (8, 16, and 32) and also different acceleration qualities (No, Low, Medium, and High). As input, we have used several well-known sequences [227] previously presented in Section 3.3.1, so in this way, we can compare our work with previous published works, and knowing how good is our work. Presented results show for each macroblock the motion from the reference frame, and the cost expressed according to the error metric, SAD (Sum of Absolutes Differences) or MSE (Mean Squared Error), although this last one is less conservative.

### *5.4.1. Results measured in KPPS (Kilo Pixels Per Second).*

KPPS is used as throughput measure for the whole system, using the SAD error metric during the execution of the algorithms. As mentioned before, we have implemented different window search sizes (8, 16, and 32) for three different algorithms (FST, 2DLOG, and TSST) using three different acceleration qualities (No, Low,

Medium, and High) in three different Nios II processor architectures (Economic (Nios II/e), Standard (Nios II/s), and Fast (Nios II/f)). In Figure 5.11, we present the achieved results using KPPS.

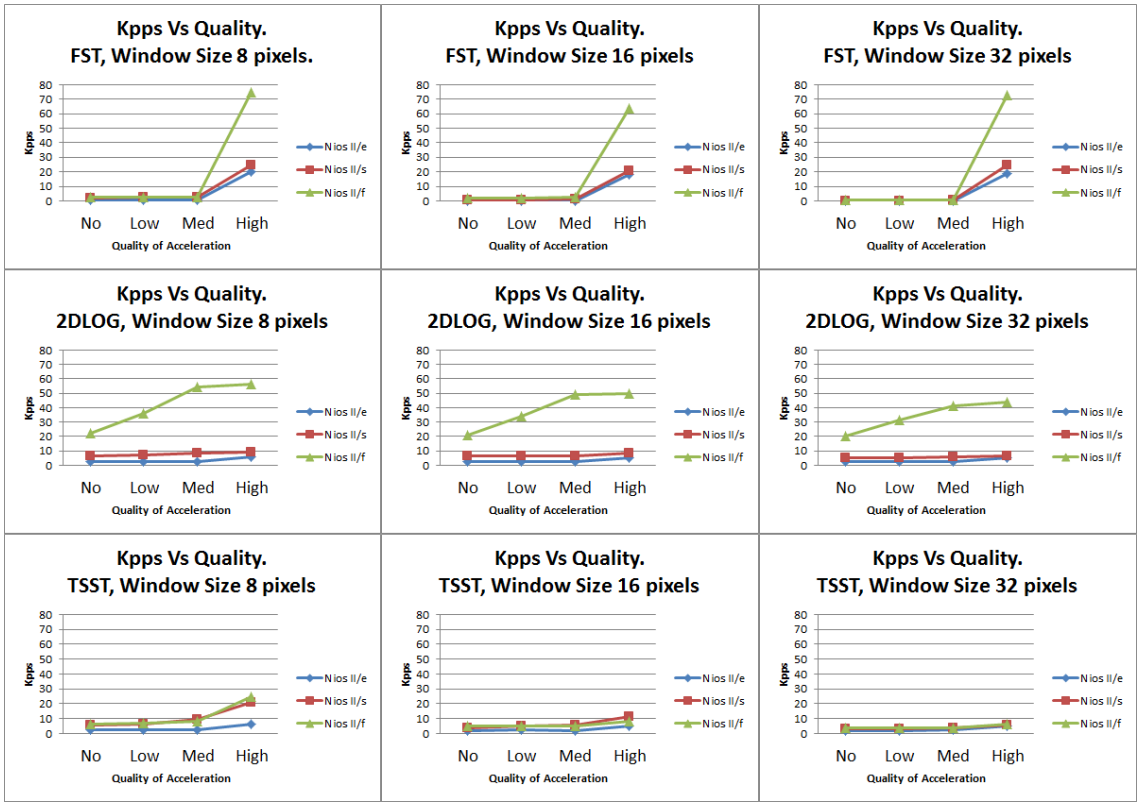


*Figure 5.11: Throughput measured in KPPS [297].*

The throughput of the whole system with Low or Medium acceleration behaves similarly when compared to the execution of the No acceleration system using the FST technique. If we focus on the TSST algorithm, this behavior becomes linear (when considering No, Low, and Medium acceleration). If we look at the 2DLOG technique, linear response is clearly seen again for the No, Low, and Medium accelerations, specially in the fast architecture (Nios II/f). Although the throughput of every architecture depends on the window size (8, 16, and 32 pixels), the linear tendency of the responses shows up for all sizes.

For instance, we notice that by using Medium acceleration on the 2DLOG technique under the Nios II/f architecture is obtained as throughput range between 16 and 21 fps (frames per second) at a 50×50 pixels resolution when using different windows sizes, and a higher throughput when using High acceleration on each selected window size.

Focusing on FST under the Nios II/f architecture, a range from 61 to 72 KPPS is delivered, depending on the window size used (from 8 to 32 pixels). This corresponds to a throughput for the system between 24.5 and 29 fps at a 50×50 pixel resolution, enough for a small sensor camera. For configurations using Nios II/e or Nios II/s, we obtain a throughput range between 20 and 27 KPPS (from 8 to 32 pixels), which corresponds to a range between 8 and 11 fps at 50×50 pixel resolution.

Regarding TSST under the Nios II/f architecture, a range from 6.15 to 24.6 KPPS is delivered, depending of the window size used (from 8 to 32 pixels). This means a throughput for the system between 2 and 10 fps at a 50×50 pixel resolution. For configurations using Nios II/e or Nios II/s, we obtain a throughput range between 5 and 20 KPPS (from 8 to 32 pixels) which means a range between 2 and 8 fps at a 50×50 pixel resolution.

The 2DLOG technique processes a range from 43.8 to 56.4 KPPS under the Nios II/f architecture, again depending on the window size used (from 8 to 32 pixels), which means a range of 17.5 – 22.5 fps. For configurations under the Nios II/e and Nios II/s architectures, we obtain a throughput of approximately 8 KPPS and 10 KPPS, independent of the window range (from 8 to 32 pixels), which means a range between 2 and 8 fps at a 50×50 pixel resolution.

Note that the size of the window is not always inversely proportional to the system throughput. For example, the TSST restricts the calculation complexity by limiting the exhaustive search to three steps, so accelerating all functions means a trade-off solution between pixel parallel level, and the increment of the macroblock size involves less macroblocks and macroblock parallel level.

### *5.4.2. Results measured in PSNR (Peak Signal to Noise Ratio).*

PSNR is used as an accuracy measure for the system using the MSE (Mean Squared Error) instead of SAD, due to the fact that MSE is more conservative and emphasize larger differences. In Figure 5.12, we present results obtained for sequences “Caltrain”, “Garden”, and “Football”, with a resolution of 352×240 (4:2:0 and CIF format).

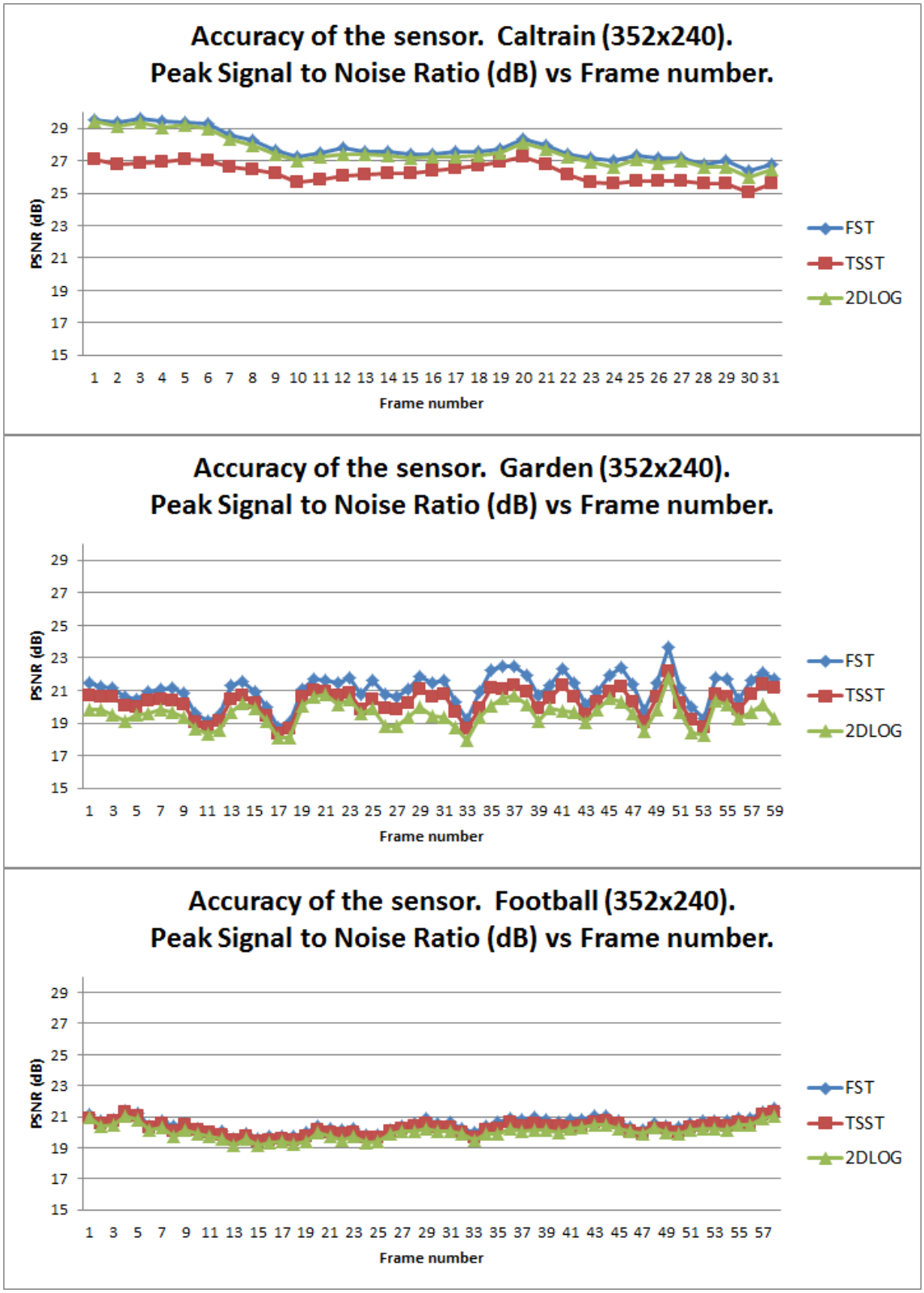


*Figure 5.12: Accuracy measured in PSNR [297].*

The accuracy of the implemented FST remains the highest of the three presented algorithms due to its exhaustive search, while the other two techniques, TSST and

2DLOG, alternate in terms of accuracy. The achieved results show a difference that is less than 2dB (decibel) between the three presented implementations on each sequence.

### *5.4.3. Results measured in used hardware resources.*

To measure the used hardware resources in our designed systems, we present the number of Logic Cells used, the number of EMs (Embedded Multiplier), and the number of Total Memory Bits in every design.

In Table 5.7, it is shown the used hardware resources for each technique (FST, 2DLOG, and TSST), fixing a window size of 32 pixels and the acceleration quality to High, under the three different processors: Nios II/e, Nios II/s and Nios II/f. Window size is fixed to 32 because it is the most expensive and presents the slowest time measures. The acceleration quality has been fixed to High because it spends more hardware resources than the other qualities.

| | | | | | | | **(FST,TSST,2DLOG)** | |
|---|---|---|---|---|---|---|---|---|
| **Nios II/ Quality** | **Method** | **Logic Cells** | **Method** | **Logic Cells** | **Method** | **Logic Cells** | **EMs(9× 9)** | **Total memory bits** |
| e / High | | 11637 (35%) | | 13173 (40%) | | 13056 (39%) | 23 (33%) | 44032 (9%) |
| s / High | FST | 12382 (37%) | TSST | 14023 (42%) | 2DLOG | 13955 (42%) | 27 (39%) | 79488 (16%) |
| f / High | | 13090 (39%) | | 14755 (44%) | | 14678 (44%) | 27 (39%) | 114944 (24%) |

*Table 5.7: Used hardware resources using window size 32[20] [297].*

As can be seen in Table 5.7, High quality acceleration requires a little bit less than 50% of the available logic cells (35% – 39% for FST, 40% – 44% for TSST, and 39% – 44% for 2DLOG). In this case, between 33% and 39% of Embedded Multipliers (9 × 9) are used, and Total Memory Bits (Block Rams) consumed are from 9% to 24%, depending on the motion estimation technique considered.

Following, it is shown used hardware resources for either FST, 2DLOG, or TSST, fixing the window size to 32 pixels, under the three different processors: Nios II/e, Nios II/s, and Nios II/f. We combine these configurations with the other three presented acceleration qualities, No, Low, and Medium.

[20] Compiled and executed with Nios II IDE v9.1sp2 and Quartus II v9.1sp2.

| Quality | Nios II/e | | | Nios II/s | | | Nios II/f | | |
|---|---|---|---|---|---|---|---|---|---|
| | **Logic Cells** | **EMs (9 × 9)** | **Total memory bits** | **Logic Cells** | **EMs (9 × 9)** | **Total memory bits** | **Logic Cells** | **EMs (9 × 9)** | **Total memory bits** |
| No | 2107 (6%) | 0 (0%) | 44032 (9%) | 3085 (9%) | 4 (6%) | 79488 (16%) | 3763 (11%) | 4 (6%) | 114944 (24%) |
| Low | 3363 (10%) | 0 (0%) | 44032 (9%) | 4147 (12%) | 4 (6%) | 79488 (16%) | 4889 (15%) | 4 (6%) | 114944 (24%) |
| Medium | 5006 (15%) | 12 (17%) | 44032 (9%) | 5812 (17%) | 16 (23%) | 79488 (16%) | 6524 (20%) | 16 (23%) | 114944 (24%) |

*Table 5.8: Used hardware resources using window size 32 [297].*

Regarding No, Low, and Medium acceleration qualities, note that the same resources are required for the three motion estimation techniques, although it depends on the processor configuration selected.

With Medium quality, we obtain an increment from 15% to 20% of Logic Cells from Nios II/e to Nios II/f. Regarding the multipliers, this increment is from 17% to 23%. Finally, regarding the Total Memory Bits consumed, this increment goes from 9% to 24.

With Low acceleration quality, the resources used are 10% – 15% (Logic Cells), 0% – 6% (Embedded Multipliers), and 9% – 24% (Total Memory bits). Regarding the No acceleration quality, we obtain an increment of 6% – 11% (Logic Cells), 0% – 6% (Embedded Multipliers), and 9% – 24% (Total Memory bits). These two increments are the same as the ones obtained for the Low acceleration quality; in other words, constant EMs and Total Memory Bits from the previous configuration are maintained.

After these conclusions, we will compare used hardware resources against resulting performance.

For this, in Figure 5.13 it is shown KPPS versus Logic Elements for every Nios II processor and motion estimation technique. Moreover it is presented using every quality acceleration (No, Low, Medium, and High) and every window size (8, 16, and 32).

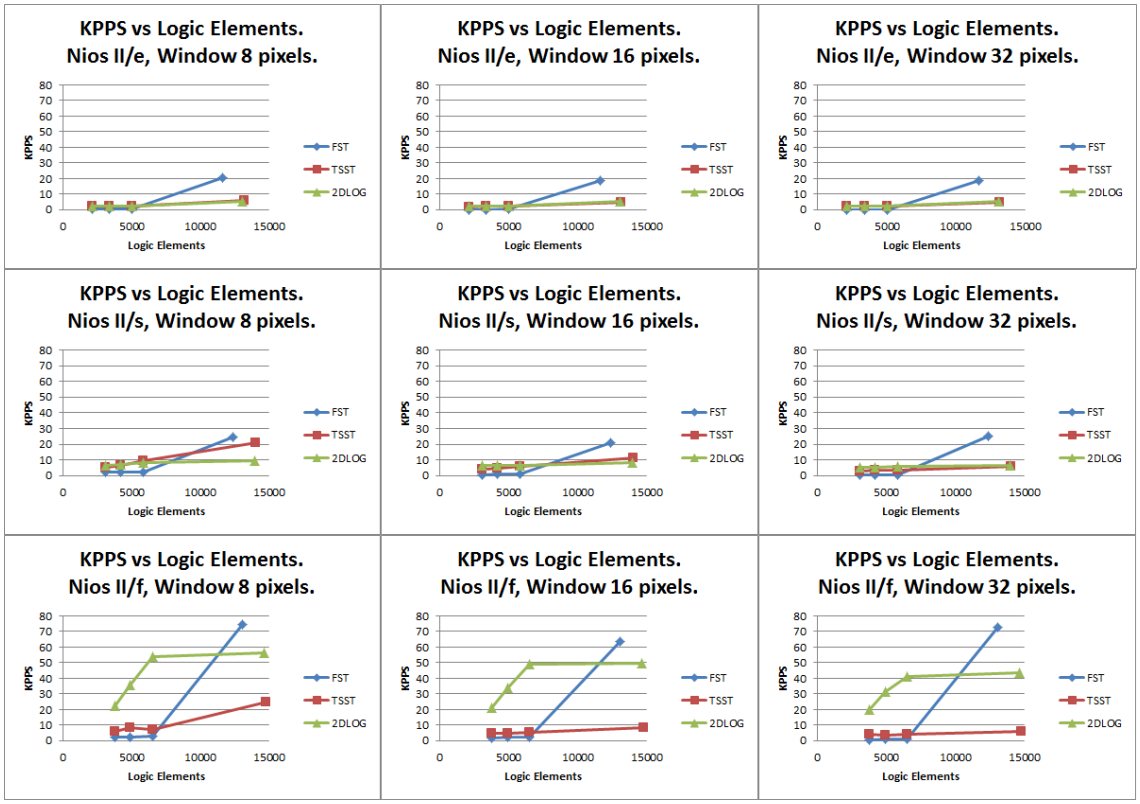


*Figure 5.13: KPPS versus Logic Elements [297].*

We can observe that, for Nios II/e, Nios II/s and Nios II/f, the FST algorithm achieves less KPPS with the same Logic Elements than TSST or 2DLOG for all acceleration qualities, except in the case of High acceleration, in which FST achieves better performance than any other using less logic elements.

Regarding the designs using the Nios II/f processor, we can see that 2DLOG gets the best performance and the TSST achieves better performance than the FST with all acceleration types, except High acceleration. In this last case, FST obtains the best results using less logic elements, followed by 2DLOG, and finally by TSST.

In Figure 5.14, it is shown KPPS versus Embedded Multipliers for every Nios II processor and motion estimation technique. It is presented using every quality acceleration (No, Low, Medium, and High) and every window size (8, 16, and 32).

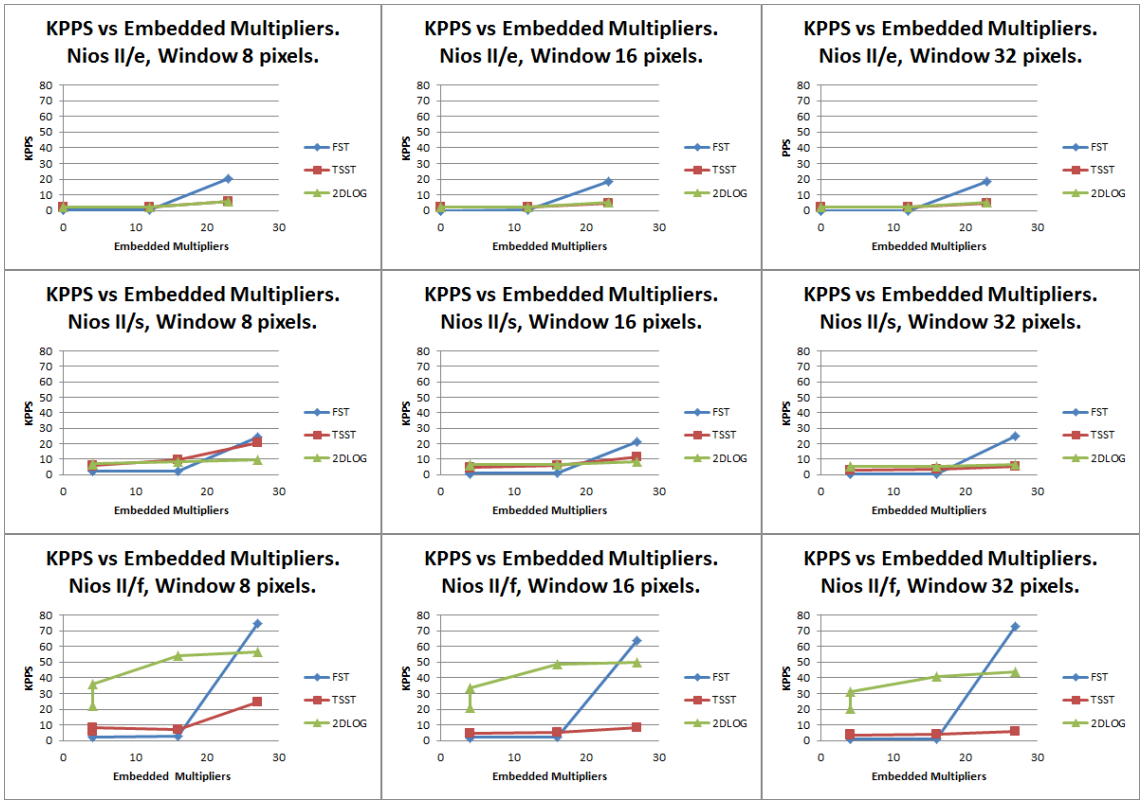


*Figure 5.14: KPPS versus Embedded Multipliers [297].*

Using the Nios II/e processor, No and Low acceleration qualities on a desired design achieves the same results, spending no Embedded Multipliers. When chosen acceleration quality is High, all algorithms use the same amount of Embedded Multipliers, although FST achieves the best performance.

When the Nios II/s processor is selected, the FST gets the worst results with No, Low, or Medium quality modes; but when acceleration is in High quality mode, the FST gets the best results using the same resources. Using this processor, the TSST gets better results than the 2DLOG in all cases except on High quality acceleration mode with a window size of 32. As the window size increases, the TSST decreases its difference against 2DLOG.

As we can observe, the FST and 2DLOG are the best designs under the Nios II/f processor, but 2DLOG gets better results using No, Low, or Medium acceleration quality modes, and FST achieves better results using the High quality acceleration mode. The TSST only can be compared with the FST in No, Low, and Medium

acceleration quality modes, where it gets similar results, while TSST achieves worse results than 2DLOG in every case.

## 5.5. Comparisons against previous works.

In this section, we first show some visual results delivered by the platform, and then we compare them to the state of the art.

We can see an example of two frames from the "Caltrain" sequence [227] corresponding to 352×240 pixels (CIF format), as shown in Figure 5.15, where the yellow arrows show the motion estimation superposed within the reference frame. If we just apply motion compensation, it is trivial to subtract the motion vectors from the current frame and transmit only the motion difference, emulating the MPEG/H.26x compression process flow, as indicated in Chapter 3.

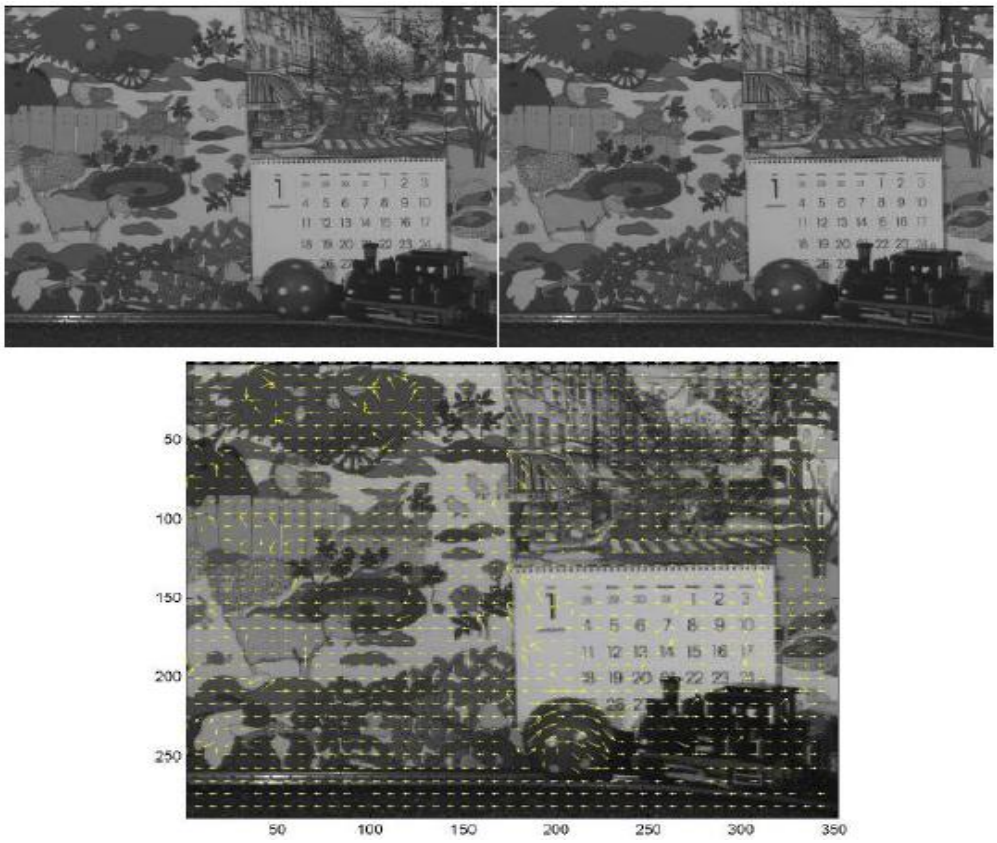


*Figure 5.15: Two consecutive "Caltrain" sequence frames (top). Motion estimation using this work FST implementation (bottom) [297].*

In the context of real-time computing sensors, there are other platforms (commented upon in Chapter 2), where families, chips used, and performance results are going to be represented. For a better knowledge of how this part of our work fits in the state of the art, we compare our throughput results with other previous works in KPPS, and we show them in the next table.

| **Models** | **Family** | **Chip used** | **Throughput (KPPS)** | **Image Size (pixels)** | **Image Rate (frame/s)** |
|---|---|---|---|---|---|
| Present work | Matching | Altera Cyclone II EP2C35F672C6/ NIOS II | 72.5 | 50×50 | 29 |
| Deutchmann [1] *et al.* [309] (1998) | Gradient H&S [34] | Full Custom VLSI | 0.12 | 20 | 5 |
| Stocker [2] *et al.* [205][310](2006) | Gradient H&S [34] | Full Custom VLSI | 5.1 | 30×30 | 6 |
| Niitsuma *et al.* [311] (2006) | Matching | Xilinx XC2V6000 | 9200 | 640×480 | 30 |
| Yong Lee *et al.* [312] (2008) | Matching | Altera Cyclone EP1C20F400C7/ NIOS II | NP | NP | NP |
| Guzman *et al.* [185] (2010) | Gradient L&K [71] | NIOS II / Eye-RIS [313] | 729 | 176×144 | 28.8 |

[1] Considering a pixel size 147 μm × 270 μm and maximum rotational velocity of 353 rpm.

[2] Taking into account a maximum Bias = 0.67 Volts.

*Table 5.9: Throughput comparisons against previous works [297].*

Although we compare our work with previous works which calculate motion estimation too, some differ in the algorithms used, and others in the chip used to implement these algorithms.

Regarding the algorithms, on the one hand we have the differential or gradient methods derived from work using the image intensity in space and time. In these methods the speed is obtained as a ratio of the above measures [112][313 – 314].

On the other hand, we have correlation based methods, frequently used in this contribution. These methods work by comparing positions from the image structure between adjacent frames and inferring the speed of the change in each location, probably the most intuitive methods [253][315 – 316].

Regarding the used chip, the approaches considered include:

- ASIC: as a methodology to design integrated circuits by specifying the layout of each individual transistor and the interconnections between them [263].

- Altera Cyclone and Cyclone II: as 130 nm and 90 nm FPGAs, to provide a customer defined feature set for high volume, cost sensitive applications [115].

- XC2V6000: a Xilinx 150 nm FPGA [317] and Nios II [318].

# Chapter VI

## Accelerating through custom instructions.

In this chapter, we describe how to accelerate our sensor application thanks to the inclusion of a new customized instruction in the Nios II processor Instruction Set Architecture (ISA). This instruction helps to reduce the time leak discovered with the profiling information in the aforementioned motion estimation techniques.

Our system has been provided with two customized instructions, which best fit to our application. A set of monocycle and multi-cycle customized instructions have been included, achieving as result a 450 KPPS performance in the best case, equivalent to a SoC which processes 50×50 @ 180 fps.

Moreover, in this chapter it is explained in detail every memory type available in our Altera FPGA platform. Extracting the most powerful memory types and doing an exhaustive testing plan with the selected memories, we have found another way to improve the system. Finally, combining the memory system design with the customized instructions explained, an additional improvement is achieved.

## 6.1. Topology and architecture description of the accelerator

As has been described previously along this work, we can accelerate the slower parts of our application with the incorporation of custom instructions into the Nios II instruction set, which can contain up to 256 instructions. A custom instruction is a user defined instruction implemented in hardware inside the processor structure, which performs the operations defined by the user and is added to the Instruction Set Architecture (ISA).

These custom instructions help us reducing a sequence of instructions from the Nios II ISA into a unique user defined instruction implemented in hardware. In Figure 6.1, it is shown how the custom instruction implementation works in parallel with the Nios II processor ALU inside the Nios II processor datapath. Using these custom instructions we can adapt the Nios II embedded processor to achieve our particular goals and a higher performance.

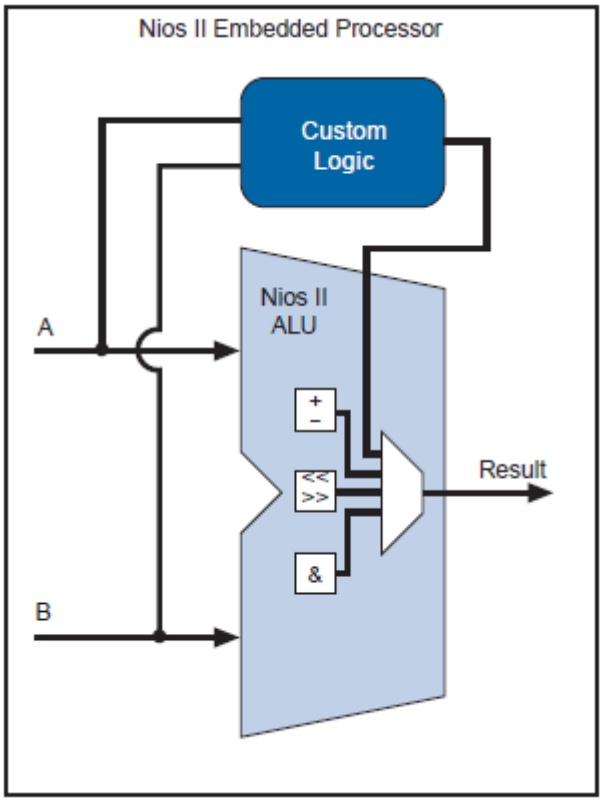


*Figure 6.1: Custom logic connected to Nios II ALU [319].*

Once the custom instruction has been added to the Nios II processor datapath, we can instantiate it directly through the Nios II processor assembly language, or directly in our source code through macros generated by Nios II EDS (Embedded Design Suite). To achieve the latter, it is used GCC (GNU Compiler Collection) built in functions, which map directly software source code to designed custom instruction.

### 6.1.1. Custom instructions integration and taxonomy.

The Nios II processor allows five different custom instruction types, providing the designer with different manners to fit the custom instruction designed to the problem at hand. In Figure 6.2, it is shown the additional ports of the different custom instruction types, that are provided to take the inputs and give the output.

Apart from the four presented custom instruction types above there is a fifth one, "External Interface", which provides the designer with the ability to interface with logic outside the Nios II processor datapath.

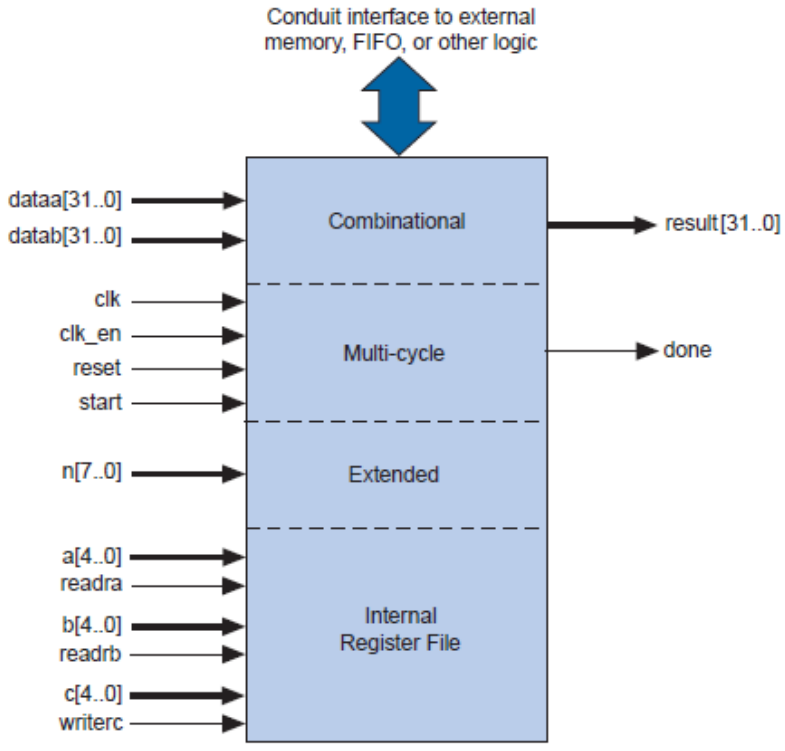


*Figure 6.2: Nios II custom instructions block diagram [319].*

Once given a brief introduction above the different custom instruction types, we present them here in a more detailed manner:

- Combinatorial: this custom instruction type provides a custom logic block which completes its execution in only one cycle. In Figure 6.3, we can see the inputs ("dataa" and "datab"), which are optional, and its output ("result"), which will be read in the rising edge of the following clock cycle.

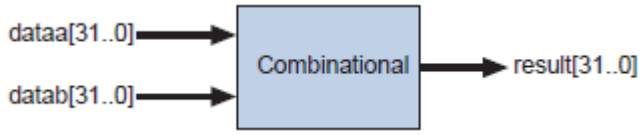


*Figure 6.3: Combinatorial custom instruction block diagram [319].*

- Multi-cycle: this custom instruction type provides a custom logic block which needs more than one clock cycle to be completed. This number of clock cycles can be fixed or variable depending on the designer. In Figure 6.4, we can see the inputs ("dataa", "datab", "clk", "clk_en", "reset", and "start") being only "clk", "clk_en", and "reset" required, and its output ("result" and "done"), which are optional. If the number of clock cycles is fixed, the processor will wait the designed number of cycles to read "result", but if the number of cycles is variable, the processor will wait until the "done" signal is activated.

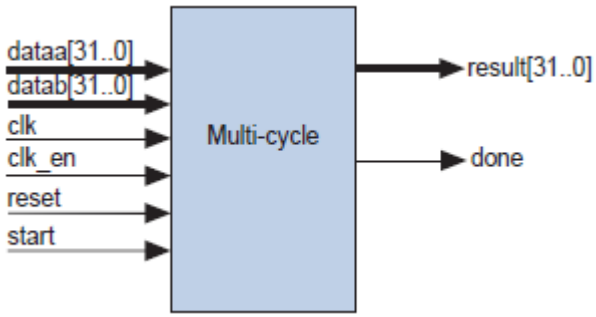


*Figure 6.4: Multi-cycle custom instruction block diagram [319].*

- Extended: this custom instruction type provides a custom logic block which has the ability to implement several operations, up to 256, selecting through an index which operation has to be executed. These custom instructions can be also combinatorial or multi-cycle, thanks to the fact that their containing operations will spend more than one custom instruction indexes. For extending a combinatorial or multi-cycle custom instruction, the designer has to add the "n" port, which has the same behavior as inputs, to the custom instruction input ports.

In Figure 6.5, it is presented an example of an extended custom instruction which has three different operations that can be performed (bit-swap, byte-swap, and half-word-swap). The desired operation is selected through the selector input ("n"), and performed over the input operand ("dataa"), for finally storing the solution in the output ("result").

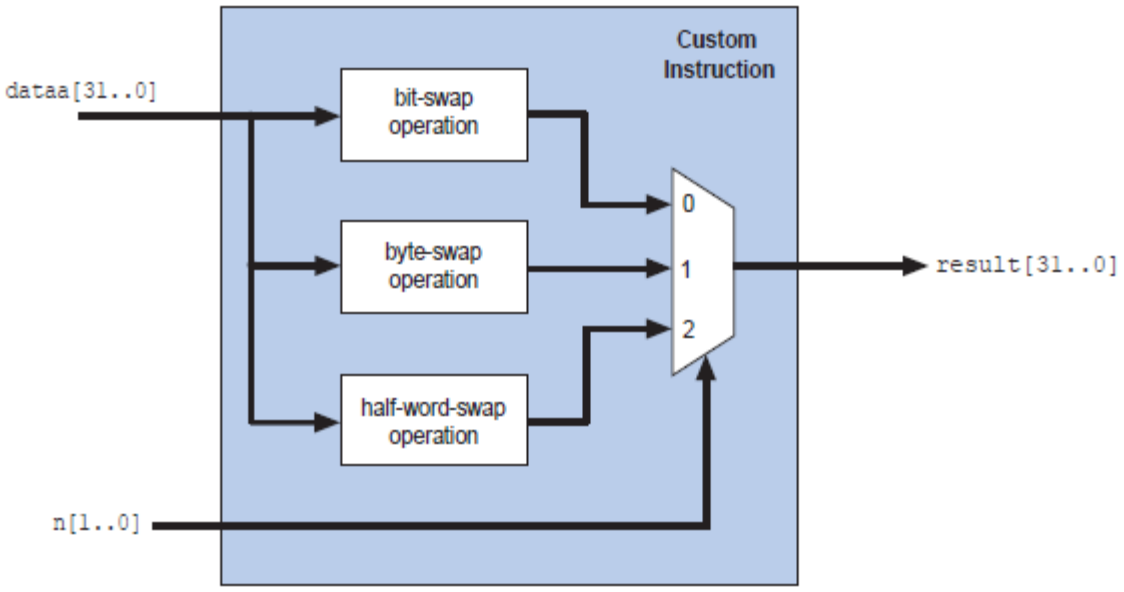


*Figure 6.5: Extended custom instruction example block diagram [319].*

- Internal register file: this custom instruction type provides a custom logic block which accesses the Nios II processor register file, allowing the designer to read the inputs and/or write the output from/to the Nios II processor register file. This custom instruction type has new inputs ("a", "b", "c", "readra", "readrb", "writerc") added to the extended custom instruction ports. With these inputs ("readra" and "readrb") the designer decides whether the custom instruction gets the inputs from the usual ports ("data" and "datab") or through the registers indexed by the new ports ("a" and "b"). The same process is done for the output with the other signal ("writerc") and the other input port ("c"). In Figure 6.6, we present an example of an internal register file custom instruction which performs an accumulation of multiplies.

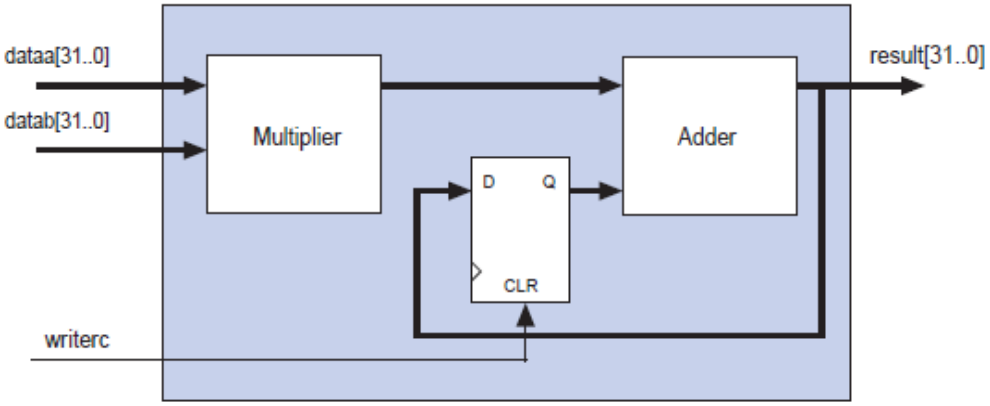


*Figure 6.6: Internal register file custom instruction example block diagram [319].*

- External interface: this custom instruction type provides a custom logic block which can interact with logic outside the Nios II processor datapath. This custom logic block inherits the other custom instruction ports, and includes also a user defined interface to connect to the logic outside the Nios II processor datapath as shown at Figure 6.7.

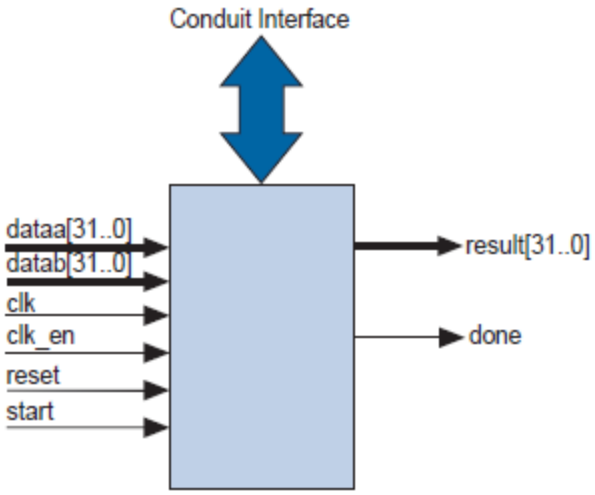


*Figure 6.7: External interface custom instruction block diagram [319].*

By examining the profiling results from Chapter 5, some conclusions can be reached about the most appropriate part of the source code to be replaced by a specific custom instruction. The GetCost function was the first in line; moreover, as shown in the tables from previous chapter, almost all the execution time was taken by it.

All the different custom instruction types, which have been presented in the present section, have been examined in order to decide which one could be the more suitable to develop a custom instruction that would replace the GetCost function source code. In our first approach it was clearly a combinatorial custom instruction, due to its speed (only one clock cycle), its easiness, and the GetCost function structure.

The multi-cycle custom instruction is also tackled in this Ph.D. Thesis, although in second place, achieving a more sophisticated custom instruction than the combinatorial one. The extended custom instruction was discarded because only one kind of operation was needed between each pair of pixels (calculate SAD). The internal register file custom instruction derives from the multi-cycle one, and that one is tackled as previously mentioned. Finally, the external interface custom instruction has been discarded due to the fact that there is no need to communicate to external interfaces.

After applying the specific designed custom instruction using the combinatorial custom instruction type, the performance enhancement for every video coding motion estimation algorithm using several testbench sequences is presented in Section 6.2.

## 6.2. Combinatorial custom instruction.

In this section, every achieved result combining every Nios II processor (Nios II/e, Nios II/s, and Nios II/f), every macroblock size (16, 32, and 64 pixels), every algorithm (FST, 2DLOG, and TSST) and every window size (8, 16, and 32 pixels), is presented when the designed combinatorial custom instruction is used, or replaced by its source code in software.

In the different subsections, the achieved results depending on the Nios II processor type, the macroblock size, or the used motion estimation technique, are described.

In Figure 6.8, the commented results for the first input, the well-known “Foreman” sequence (frames 0 and 1), are shown.

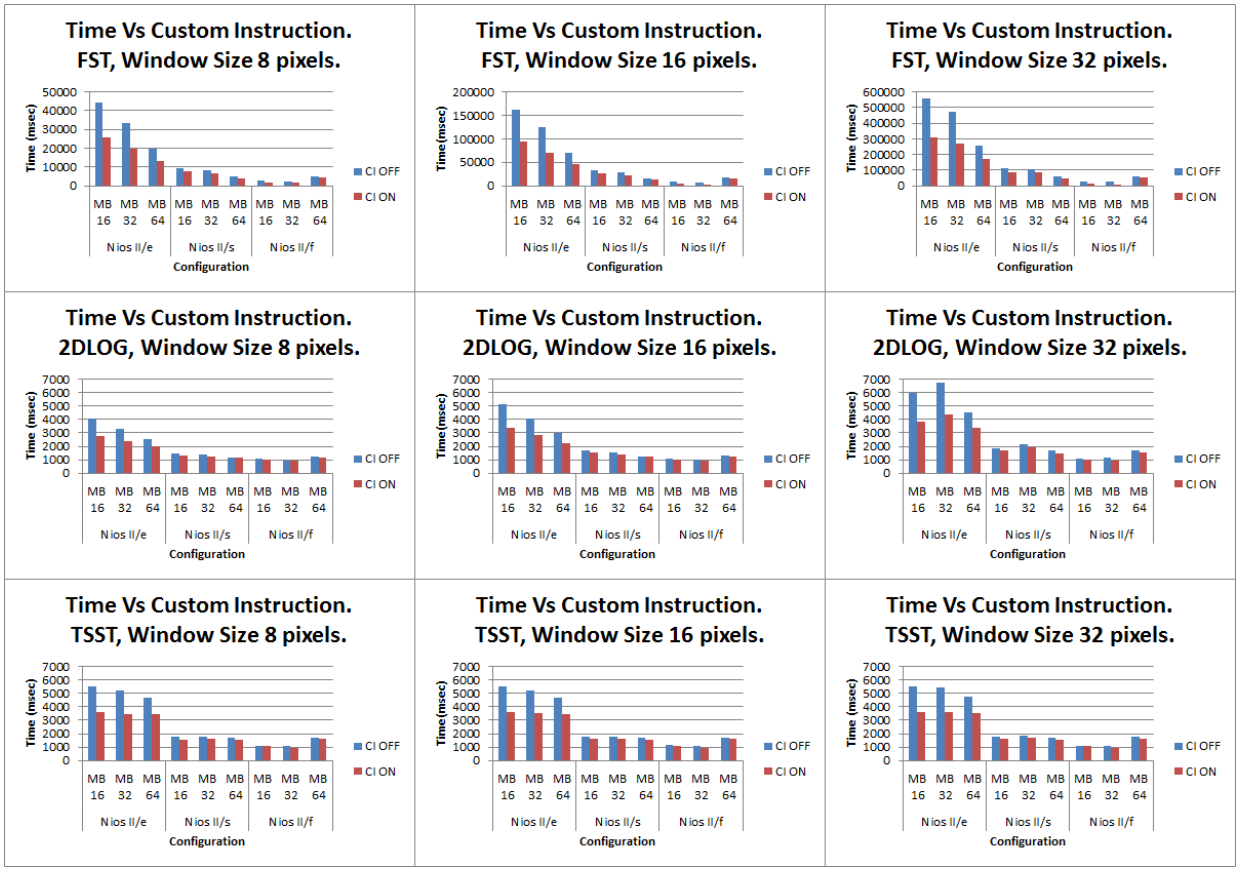


*Figure 6.8: Achieved results for "Foreman" sequence [320].*

In Table 6.1, it is shown the time reduction for Figure 6.8 results. In this way, it is possible to see at a glance how much time we are saving with the improvement using our designed custom instruction. The best case achieves an improvement of 54% when executing under the Nios II/f processor the FST algorithm using a window size of 32 and a macroblock size of 16. On the other hand, the worst case remains without improvement when executing under the Nios II/f processor the TSST algorithm using a window size of 32 and a macroblock size of 16.

| **Algorithm / Window size** | **Processor** | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | **Nios II/e** | | | **Nios II/s** | | | **Nios II/f** | | |
| | **MB 16** | **MB 32** | **MB 64** | **MB 16** | **MB 32** | **MB 64** | **MB 16** | **MB 32** | **MB 64** |
| FST/8 pixels | 41.72% | 41.88% | 32.23 % | 20.02 % | 16.73% | 15.84% | 39.74% | 36.22% | 9.13% |
| FST/16 pixels | 42.58% | 42.90% | 33.29% | 21.08% | 18.86% | 17.91% | 50.22% | 44.96% | 11.15 % |
| FST/32 pixels | 44.23% | 43.21% | 33.62% | 21.61% | 18.99% | 18.42% | 54.00% | 53.85% | 11.49% |
| 2DLOG/8 pixels | 32.35% | 27.63% | 20.63% | 8.78% | 7.25% | 4.20% | 6.36% | 5.15% | 4.13% |
| 2DLOG/16 pixels | 34.05% | 30.05% | 23.31% | 11.63% | 6.54% | 3.94% | 4.63% | 8.00% | 4.65% |
| 2DLOG/32 pixels | 35.45% | 35.46% | 25.61% | 10.11% | 10.33% | 11.24% | 5.50% | 14.91% | 5.39% |
| TSST/8 pixels | 34.97% | 33.46% | 26.18% | 11.24% | 10.56% | 8.98% | 0.93% | 11.21% | 4.71% |
| TSST/16 pixels | 34.67% | 33.14% | 26.50% | 11.11% | 9.44% | 9.52% | 4.42% | 11.21% | 4.12% |
| TSST/32 pixels | 34.37% | 33.46% | 25.79% | 9.55% | 10.22% | 8.77% | ~0.00% | 11.21 % | 7.39% |

*Table 6.1: Achieved time savings for "Foreman" sequence [320].*

Now, in Figure 6.9 the commented results for the second stimulus, the well-known "Carphone" sequence (frames 0 and 1), are shown.

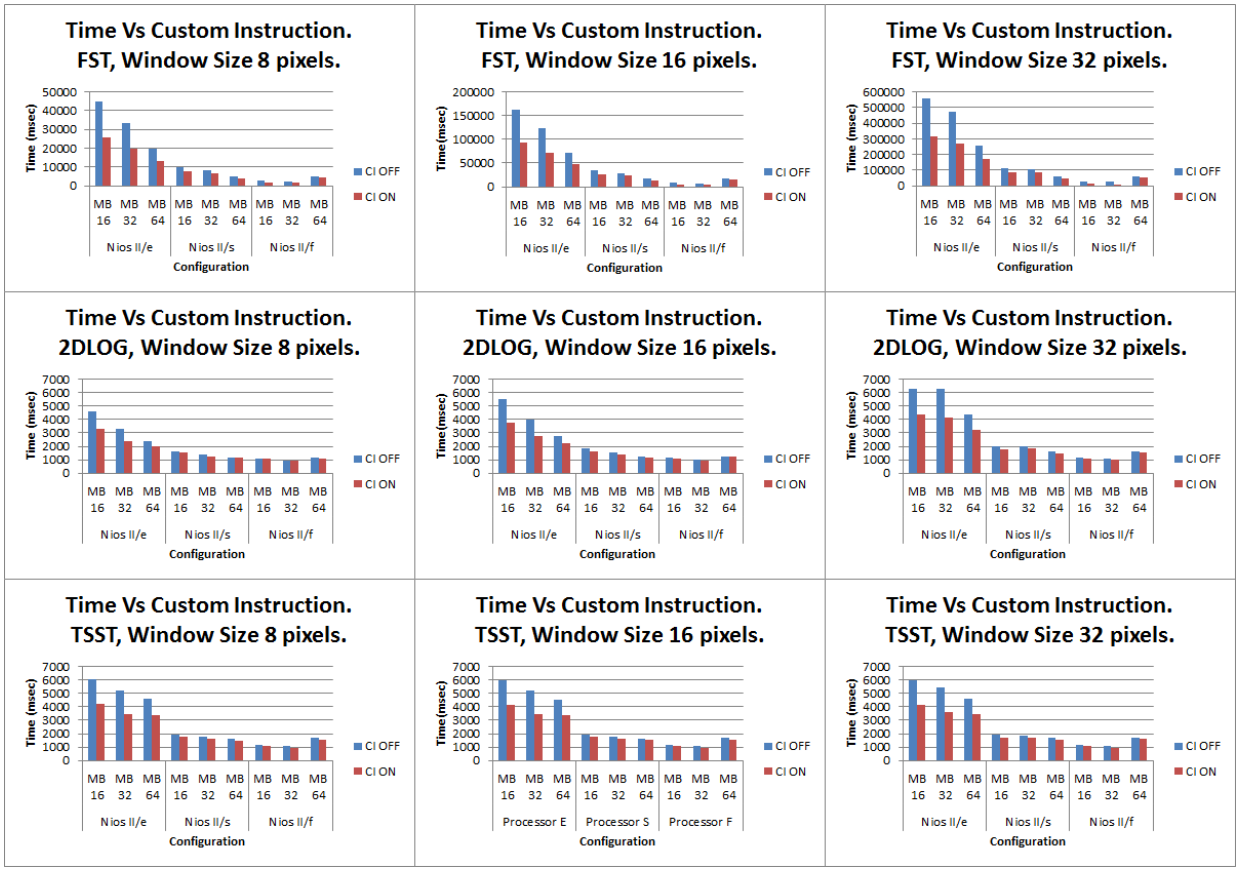


*Figure 6.9: Achieved results for "Carphone" sequence [320].*

In Table 6.2, the time reduction for Figure 6.9 results is shown. In this way, it is possible to see at a glance how much time we are saving with our improvement using our designed custom instruction. The best case achieves an improvement of 56.1% when executing under the Nios II/f processor the FST algorithm using a window size of 32 and a macroblock size of 16. On the other hand, the worst case remains without improvement when executing under the Nios II/s processor the 2DLOG algorithm using a window size of 8 and a macroblock size of 64.

| Algorithm / Window size | Processor | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | Nios II/e | | | Nios II/s | | | Nios II/f | | |
| | MB 16 | MB 32 | MB 64 | MB 16 | MB 32 | MB 64 | MB 16 | MB 32 | MB 64 |
| FST/8 pixels | 42.43% | 41.76% | 31.72% | 19.12% | 17.00% | 15.67% | 40.39% | 35.83% | 9.87% |
| FST/16 pixels | 43.58% | 42.87% | 33.24% | 21.02% | 18.57% | 17.74% | 51.73% | 49.03% | 10.98% |
| FST/32 pixels | 43.91% | 43.17% | 33.69% | 21.44% | 18.99% | 18.48% | 56.01% | 53.73% | 11.53% |
| 2DLOG/8 pixels | 28.76% | 28.53% | 17.92% | 6.79% | 7.91% | ~0.00% | 0.92% | 5.26% | 1.75% |
| 2DLOG/16 pixels | 31.65% | 30.63% | 20.22% | 10.44% | 7.84% | 5.56% | 0.88% | 6.06% | 3.17% |
| 2DLOG/32 pixels | 31.05% | 34.76% | 26.27% | 10.15% | 9.36% | 7.55% | 5.13% | 10.81% | 4.94% |
| TSST/8 pixels | 30.51% | 33.65% | 27.02% | 9.84% | 8.94% | 8.54% | 8.47% | 10.38% | 5.39% |
| TSST/16 pixels | 30.99% | 33.85% | 25.61% | 10.77% | 10.11% | 7.32% | 6.72% | 11.82% | 5.39% |
| TSST/32 pixels | 30.92% | 33.52% | 25.43% | 10.42% | 10.22% | 7.78% | 9.24% | 10.19% | 5.88% |

*Table 6.2: Achieved time savings for "Carphone" sequence [320].*

### *6.2.1. Results related to processors Nios II/e, Nios II/s, and Nios II/f.*

In this subsection, we discuss the presented results for both stimuli, considering an average between the obtained results, and focusing on the selected Nios II processor. To visualize the achieved improvements when using our custom instruction, Figure 6.10 shows the achieved results when turning our custom instruction on, for every executed algorithm, every macroblock size, and every window search size, grouped by Nios II processor type.

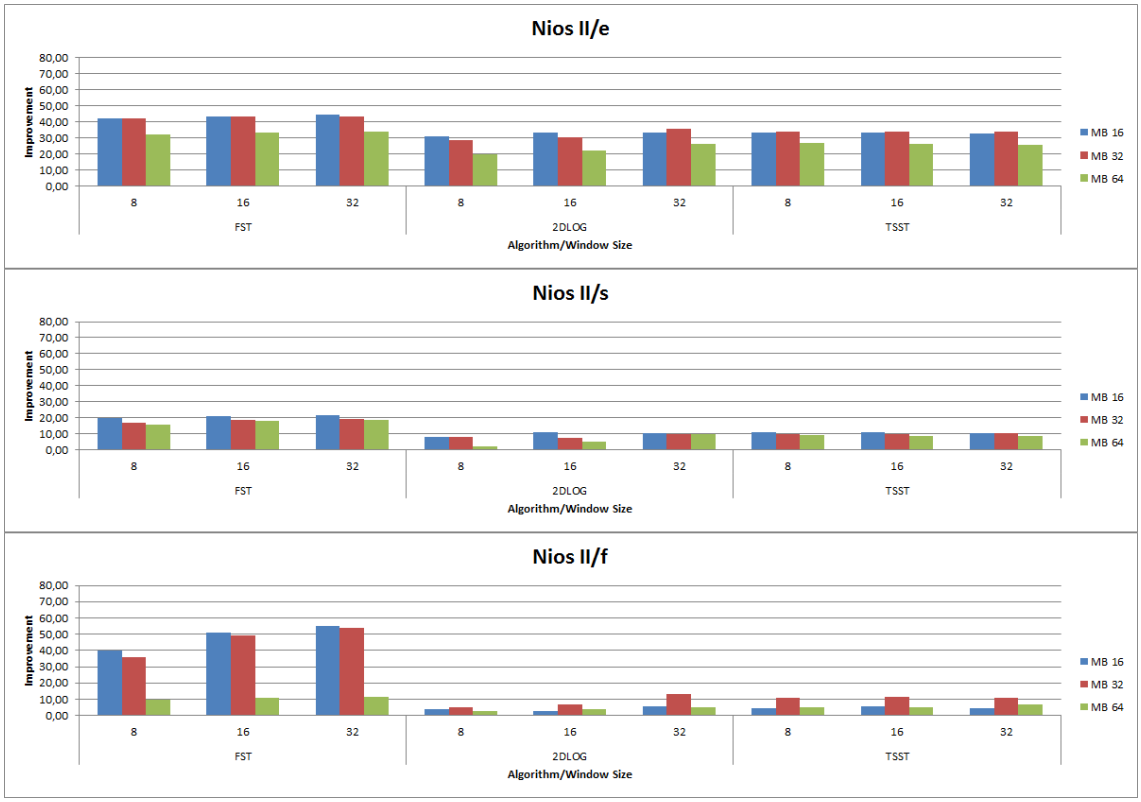


*Figure 6.10: Results focused on Nios II processor.*

As we can see above at Figure 6.10, the achieved results are more constant when working with the Nios II/e processor, due to the fact that this processor has neither data cache nor instruction cache and it is possible to achieve a high improvement on each case. Indeed, the achieved time execution saving average is more than 35% selecting 16 and 32 macroblock sizes independently of the window search size, and nearly 30% selecting 64 as macroblock size independently of the window search size.

Nios II/e processor maximum time saving is 44.23%, which is achieved executing the FST algorithm in the "Foreman" test sequence with 16 as macroblock size using a window search of 32. The minimum time saving achieved executing under the Nios II/e processor, which is 17.92%, is reached executing the 2DLOG algorithm in the "Carphone" sequence with 64 as macroblock size using a window search of 8.

The carried out averages, maximum, and minimum, make the Nios II/e processor the most suitable for achieve constant improvements.

Focusing now on the Nios II/s processor, we can also observe constant results, although they are lower compared to the Nios II/e processor, because the Nios II/s

processor contains an instruction cache which improves the execution of the selected algorithm when not using the custom instruction. Indeed, the achieved time execution saving average is between 10% and 15% in every case when grouping by macroblock size.

Nios II/s processor execution time saving goes from zero as the minimum achieved, in the case of executing the 2DLOG algorithm in the "Carphone" sequence with 64 as macroblock size using a window search of 8, to 21.61% as the maximum value, when executing the FST algorithm in the "Foreman" test sequence with 16 as macroblock size using a window search of 32.

These improvements make the Nios II/s processor not so constant as previously commented of the Nios II/e processor, but good in nearly every case due to the achieved execution time savings are around 10%.

Now, regarding the Nios II/f processor, we can see that accomplished improvements are generally lower compared to the two other processors, due to the fact that this processor has both instruction and data cache, which improve the execution of the selected algorithm when not using the custom instruction. We can also observe that, when executing the FST algorithm with macroblock sizes 16 and 32, the achieved improvements are much higher, due to the combination of the high number of the custom instruction calls with both caches. The reached time execution saving average is around 20% in every case with macroblock sizes 16 and 32. This average is between the achieved when executing in the Nios II/e processor and the achieved when executing in the Nios II/s processor. In the case of macroblock size 64, average execution time savings are between 5% and 10%, that is to say, lower than the two previous Nios II processors.

Nios II/f processor maximum time saving is the maximum among every processor, being a 56.01%, which is achieved executing the FST algorithm in the "Carphone" test sequence with 16 as macroblock size and using a window search of 32. As when executing under the Nios II/s processor, there is another case where execution time saving does not exist, which is executing the TSST algorithm in the "Foreman" sequence with 16 as macroblock size using a window search of 32.

These improvements make the Nios II/f processor the most extreme, because it can achieve both, a null execution time saving and the highest one.

### *6.2.2. Results related to macroblock sizes 16, 32, and 64.*

In this subsection, we discuss the presented results for each one of the stimuli sequences, calculating the average between the results, and focusing on the selected macroblock size. For visualizing the achieved improvements when using our custom instruction, Figure 6.11 shows the achieved results when turning our custom instruction on, for every executed algorithm, every Nios II processor, and every window search size, grouped by the macroblock size selected.

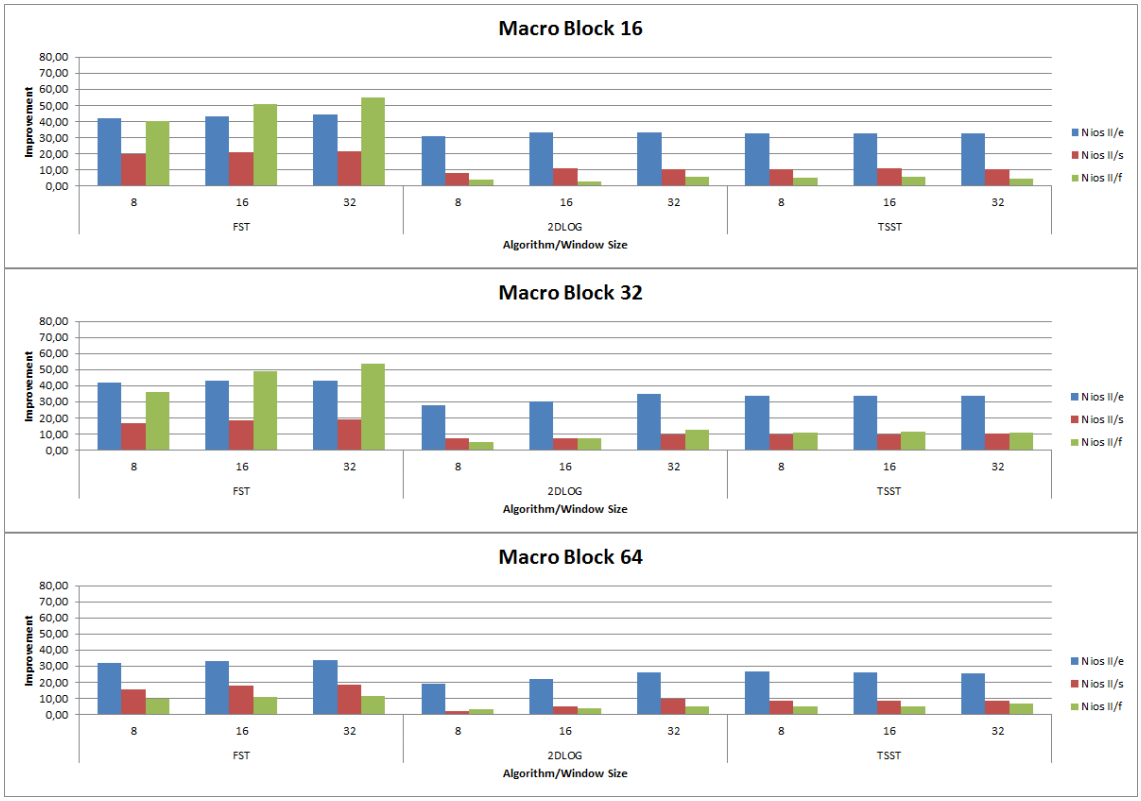


*Figure 6.11: Results focused on macroblock size.*

According to the presented results at Figure 6.11, using a macroblock size of 16, execution time savings averages go from more than 10% achieved when executing in the Nios II/s processor, to nearly 40%, which is achieved when executing in the Nios II/e processor.

With macroblock size 16, the best execution time saving, 56.01%, is achieved executing under the Nios II/f processor the FST algorithm in the "Carphone" test sequence using a window search of 32. In the opposite side, we have a null execution time saving, achieved when executing under the Nios II/f processor the TSST algorithm in the "Foreman" sequence using a window search of 32.

These results makes this macroblock size the most extreme, because achieved results using it go from the minimum achieved in any test (no improvement), to the maximum achieved in any test (56.01%), due to the fact that the custom instruction is called more times.

Now, focusing on the macroblock size of 32, results are more constant. In spite of having execution time savings averages similar to the ones achieved with macroblock size 16 (from more than 10% achieved when executing in the Nios II/s processor to nearly 40% when executing in the Nios II/e processor), its results are more similar to the average, and are also high due to the fact that the custom instruction is also called many more times than when using macroblock size of 64.

Using macroblock size 32, the maximum execution time saving achieved, 53.85%, when executing under the Nios II/f processor the FST algorithm in the "Foreman" test sequence using a window search size of 32, is smaller than the one achieved with macroblock size 16. The minimum execution time saving achieved is more similar to the average, than the one achieved when using macroblock size 16. In fact, the minimum achieved with macroblock size 32 is a 5.15%, achieved when executing under the Nios II/f processor the 2DLOG algorithm in the "Foreman" test sequence using a window search of 8.These measures make macroblock size 32 a bit more constant in execution time saving than macroblock size 16.

Regarding the macroblock size 64, achieved results are not so high as the other two macroblock sizes due to the fact that are needed less macroblocks to cover all the frame size and the custom instruction is called many less times.

With macroblock size 64, the execution time saving averages go from a bit more than 5%, when executing in the Nios II/f processor, to nearly 30%, when executing in the Nios II/e processor. Using this macroblock size, there is another case of null

improvement, which is achieved executing under the Nios II/s processor the 2DLOG algorithm in the "Carphone" test sequence using a window search of 8. As maximum, this macroblock size only achieves a 33.69%, executing under the Nios II/e processor the FST algorithm in the "Carphone" sequence for window search of 32.

These results makes macroblock size 64 the worst for achieving high improvements, but on the other hand, it results in more constant improvements than the previously presented macroblock sizes.

### *6.2.3. Results related to algorithms FST, 2DLOG, and TSST.*

In this subsection, we discuss the presented results for the testbench sequences, calculating the average between the achieved results, and focusing on the selected motion estimation technique. For visualizing the achieved improvements when using our custom instruction, Figure 6.12 shows the achieved results when turning on our custom instruction for every macroblock size, every Nios II processor, and every window search size, grouped by the selected algorithm.

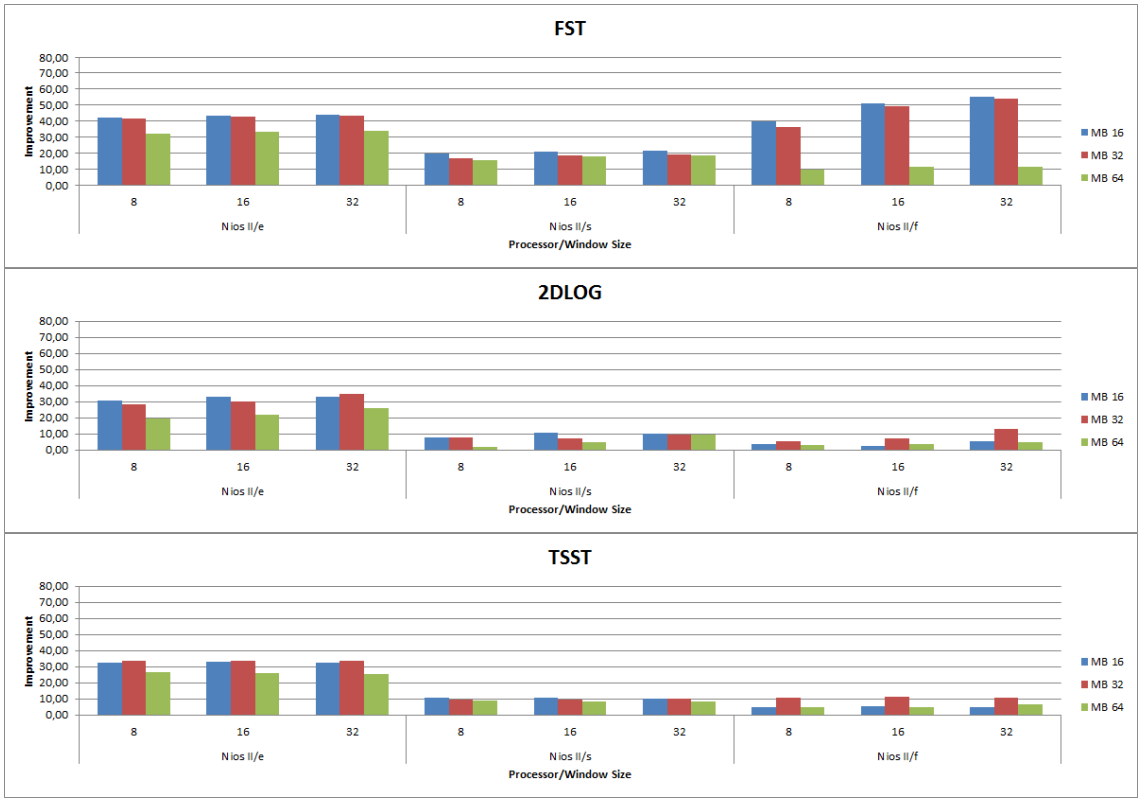


*Figure 6.12: Results focused on algorithm.*

Focusing on the FST algorithm, we can see that the execution time savings achieved are the highest of every algorithm due to the fact that it is the algorithm with more calls to the custom instruction. With the FST algorithm, execution time saving averages goes from around 20% in the case of using a macroblock size of 64, to nearly 40% in the case of using macroblock size of 16. Moreover, minimum achieved times savings go from nearly 10% in the case of executing under the Nios II/f processor using a macroblock size of 64 and a window search of 8, to nearly 20% when executing under the Nios II/s processor using a macroblock size of 16 and a window search of 8.

This suggests that the FST algorithm is the most suitable for achieving higher execution time savings. Indeed, executing this algorithm the highest execution time saving is achieved, which is the previous described 56.01%.

Regarding the 2DLOG algorithm, we can conclude that execution time savings are lower than when executing FST due to the fact that it calls many less times the designed custom instruction. This is translated in a null execution time saving when executing under the Nios II/s processor the "Carphone" sequence using a macroblock size of 64 and a window search of 8, or in an execution time saving lower than 1% in the cases of executing under the Nios II/f processor the same sequence but using a macroblock size of 16 and a window search of 8 or 16. Apart from this, the execution time saving achieved varies in average from more than 10% to a slightly higher than 15% depending on the selected macroblock size.

Focusing on the TSST algorithm, it shows execution time savings similar to the 2DLOG algorithm due to amount of calls to the custom instruction is only a bit lower. The TSST algorithm presents some low minimums, like a null execution time saving when executing under the Nios II /f processor the "Foreman" sequence with a macroblock size of 16 and a window size of 32, or a lower than 1% time saving when executing under the Nios II/f processor the same sequence with the same macroblock size but using a window size of 8.

In spite of this, execution time savings averages are more stable than when executing the 2DLOG algorithm. These averages go from more than 13% to slightly higher than 18% depending on the selected macroblock size.

Summarizing these measures, the 2DLOG algorithm is the least suitable for achieving execution time savings. The TSST algorithm is more suitable to achieve good execution time saving although not so good as the FST algorithm.

Even though, the 2DLOG algorithm achieves a higher maximum, 35.46%, when executing under the Nios II/e processor the "Foreman" sequence with a macroblock size of 32 and a window size of 32, than the one achieved executing the TSST algorithm, 34.97%, when executing under the Nios II/e processor the "Foreman" sequence with a macroblock size of 16 and a window size of 8. On the other hand, we have the FST algorithm, whose averages are not so constant like was the case with the other two algorithms but they are higher, varying from around 20% to more than 40%, due to the fact that the number of iterations of the algorithm varies notably depending on the selected parameters. Moreover, its minimums vary from nearly 10% to nearly 20%, and its maximums go from more than 30% to more than 55%, which demonstrates its potential to be improved.

## 6.3. Multi-cycle custom instruction.

In this section, we tackle the use of a custom instruction designed as a Nios II custom instruction of the multi-cycle type to replace the GetCost function source code as was pointed by the profiling done in Section 5.3.1, substituting the previous presented combinatorial custom instruction with the designed multi-cycle custom instruction.

As explained before, the GetCost function is used to calculate the SAD between two macroblocks, one from the reference frame, and the other from the current frame. When replacing GetCost source code by the combinatorial custom instruction, the latter is called for every pair of pixels, one from each macroblock, and the local SAD calculated is accumulated to give later the total SAD between the required pair of macroblocks.

On the other hand, when replacing GetCost source code by the multi-cycle custom instruction, the latter is called for every group of eight pixels, four from each macroblock, and its main feature, the multi-cycle architecture, is used to calculate an accumulated SAD for that group. In this way, executing the muti-cycle custom instruction produces little accumulated SADs, which are then added to achieve the total SAD between the required pair of macroblocks.

In Figure 6.13, we present all the achieved results for every possible combination between the executed algorithm (FST, 2DLOG, and TSST), the selected macroblock size (16, 32, and 64), and the used window size (8, 16, and 32), for the "Foreman" test sequence (frames 0 and 1) using the multi-cycle custom instruction. They will be shown in comparison with the combinatorial custom instruction case and the base case, which we consider to be the GetCost source code translated to Nios II processor instructions directly.

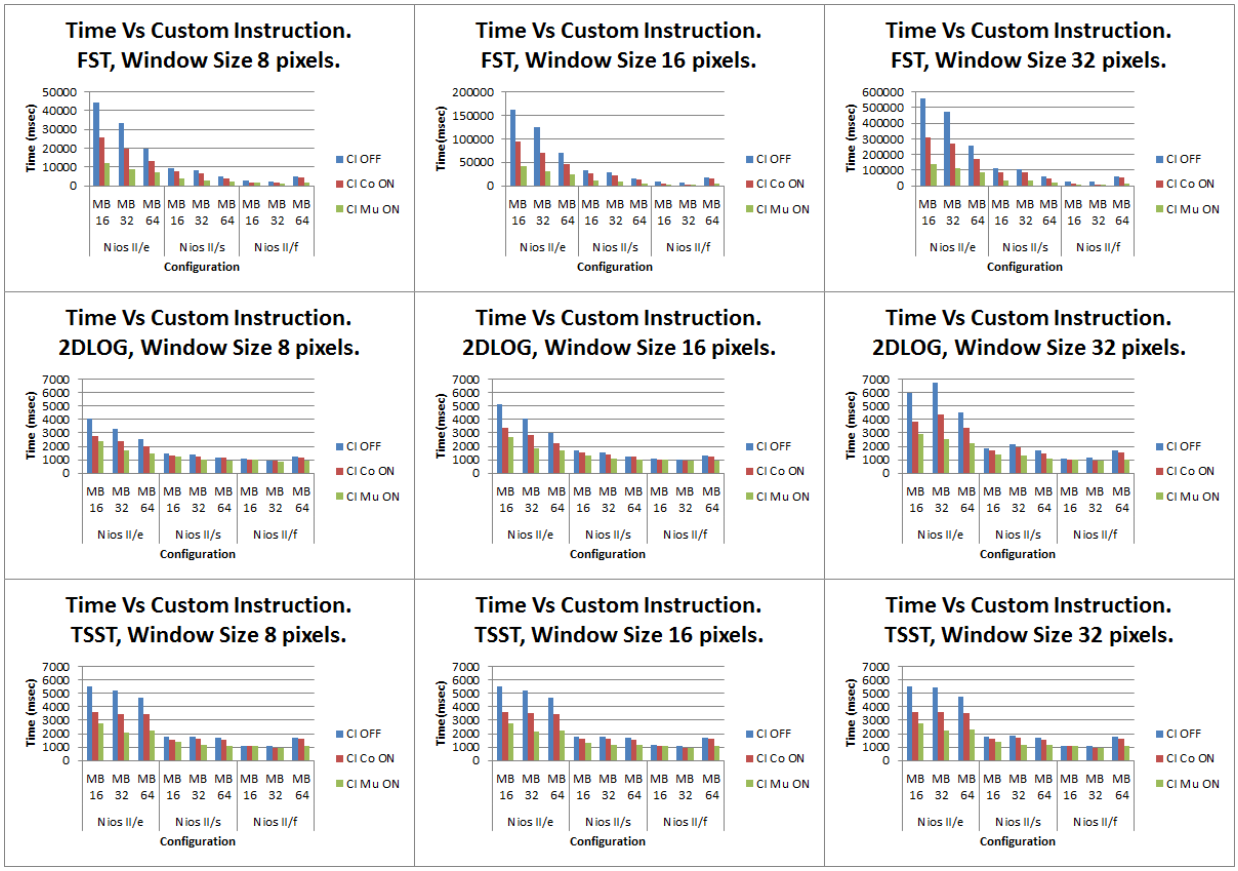


*Figure 6.13: Achieved results for "Foreman" sequence.*

In Table 6.3, it is shown the time reduction for Figure 6.13 results. In this way, it is possible to analyze at a glance, how much time we are saving with our improvement using our designed custom instruction. The best case achieves an improvement of 76.08% when executing under the Nios II/e processor the FST algorithm using a window search of 32 and a macroblock size of 32. On the other hand, the worst case remains without improvement when executing under the Nios II/f processor the TSST algorithm using a window size of 32 and a macroblock size of 16.

| **Algorithm / Window size** | **Processor** | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | **Nios II/e** | | | **Nios II/s** | | | **Nios II/f** | | |
| | **MB 16** | **MB 32** | **MB 64** | **MB 16** | **MB 32** | **MB 64** | **MB 16** | **MB 32** | **MB 64** |
| FST/8 pixels | 72.10% | 73.52% | 63.75% | 60.38% | 60.97% | 57.23% | 48.34% | 45.28% | 62.17% |
| FST/16 pixels | 74.52% | 75.53% | 66.39% | 65.94% | 66.39% | 64.96% | 62.81% | 60.03% | 70.14% |
| FST/32 pixels | 75.16% | 76.08% | 67.11% | 67.54% | 67.96% | 67.42% | 67.61% | 65.65% | 72.59% |
| 2DLOG/8 pixels | 40.49% | 49.85% | 40.48% | 16.22% | 24.64% | 20.17% | 8.18% | 9.28% | 23.97% |
| 2DLOG/16 pixels | 46.77% | 54.68% | 42.23% | 22.67% | 28.76% | 22.05% | 4.63% | 10.00% | 25.58% |
| 2DLOG/32 pixels | 50.84% | 62.61% | 51.00% | 25.00% | 39.44% | 34.32% | 6.42% | 17.54% | 37.13% |
| TSST/8 pixels | 50.27% | 59.19% | 52.36% | 23.60% | 35.00% | 34.13% | 0.93% | 14.95% | 37.06% |
| TSST/16 pixels | 49.45% | 58.43% | 52.35% | 25.56% | 33.89% | 32.74% | 6.19% | 13.08% | 37.65% |
| TSST/32 pixels | 48.99% | 59.38% | 51.57% | 23.03% | 37.10% | 33.33% | 0.00% | 14.95% | 38.64% |

*Table 6.3: Achieved time savings for "Foreman" sequence.*

Now, in Figure 6.14 the commented results for the second input, the well-known "Carphone" sequence (frames 0 and 1), are shown.

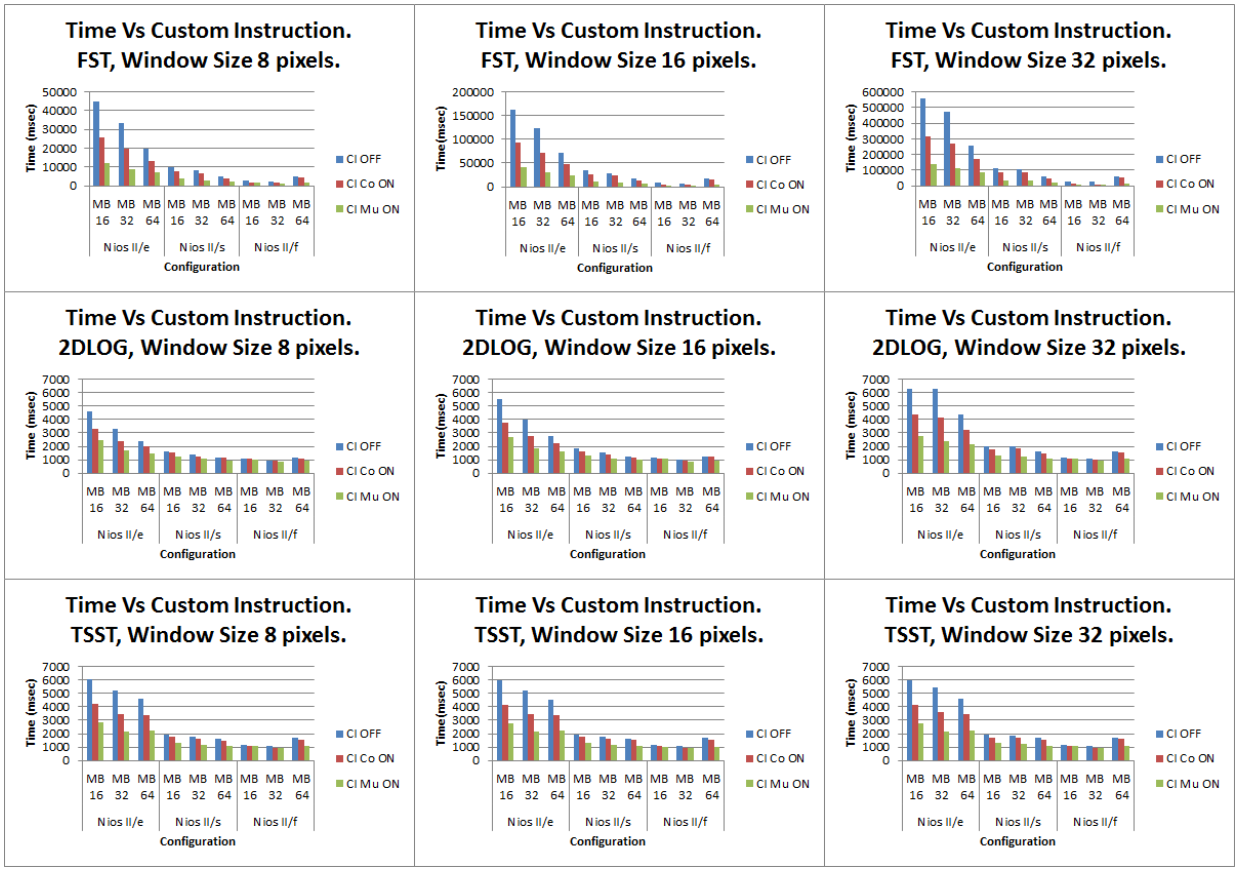


*Figure 6.14: Achieved results for "Carphone" sequence.*

In Table 6.4, it is shown the time reduction for Figure 6.14 results. Again, in this way, it is possible to see at a glance how much time we save with our improvement using the remarked custom instruction.

The best case achieves an improvement of 76.07% when executing under the Nios II/e processor the FST algorithm using a window search of 32 and a macroblock size of 32.

On the other hand, the worst case achieves and improvement of 6.19% when executing under the Nios II/f processor the 2DLOG algorithm using a window search of 16 and a macroblock size of 16.

| Algorithm / Window size | Processor | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | Nios II/e | | | Nios II/s | | | Nios II/f | | |
| | MB 16 | MB 32 | MB 64 | MB 16 | MB 32 | MB 64 | MB 16 | MB 32 | MB 64 |
| FST/8 pixels | 72.35% | 73.41% | 63.69% | 60.12% | 60.79% | 56.94% | 49.51% | 45.67% | 61.86% |
| FST/16 pixels | 74.61% | 75.55% | 66.42% | 66.30% | 66.31% | 64.94% | 62.54% | 59.97% | 70.01% |
| FST/32 pixels | 75.16% | 76.07% | 67.15% | 67.80% | 67.95% | 67.42% | 67.78% | 65.67% | 72.60% |
| 2DLOG/8 pixels | 45.97% | 49.25% | 38.75% | 22.22% | 21.58% | 17.24% | 6.42% | 7.37% | 20.18% |
| 2DLOG/16 pixels | 51.54% | 52.66% | 40.79% | 28.57% | 29.41% | 20.63% | 6.19% | 11.11% | 26.19% |
| 2DLOG/32 pixels | 55.57% | 62.06% | 50.69% | 31.47% | 38.42% | 32.08% | 6.84% | 14.41% | 34.57% |
| TSST/8 pixels | 53.40% | 59.04% | 51.63% | 30.05% | 34.64% | 32.32% | 10.17% | 14.15% | 36.53% |
| TSST/16 pixels | 54.27% | 58.46% | 50.99% | 30.77% | 33.71% | 31.71% | 11.76% | 16.36% | 37.13% |
| TSST/32 pixels | 54.12% | 60.07% | 51.52% | 31.25% | 34.95% | 32.93% | 10.92% | 15.74% | 37.06% |

*Table 6.4: Achieved time savings for "Carphone" sequence*

### 6.3.1. Results related to processors Nios II/e, Nios II/s, and Nios II/f.

In this subsection, we discuss the presented results for both sequences, calculating the statistical average between the results, and focusing on the selected Nios II processor. Following the previous methodology to visualize the achieved improvements when using our custom instruction, Figure 6.15 shows the achieved results when turning on our custom instruction for every executed algorithm, every macroblock size, and every window search size, grouped by Nios II processor type.

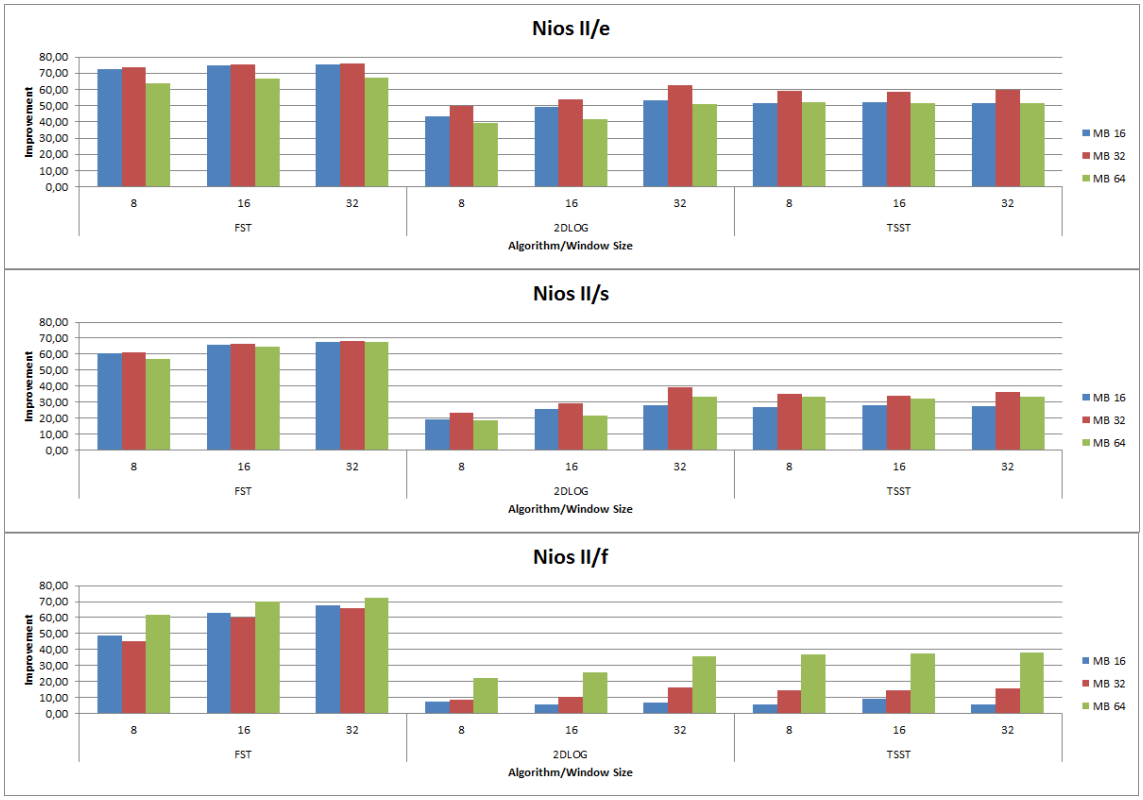


*Figure 6.15: Results focused on Nios II processor.*

As we can see, the achieved results are more constant when working with the Nios II/e processor, due to the fact that this processor has neither data cache nor instruction cache and it is possible to achieve a high improvement on each case. Indeed, the achieved time execution saving average is more than 60% selecting 16 and 32 macroblock sizes independently of the window search size, and nearly 55% selecting 64 as macroblock size independently of the window search size.

Nios II/e processor maximum time saving is 76.08%, which is achieved executing the FST algorithm in the "Foreman" test sequence with 32 as macroblock size using a window search of 32. The minimum time saving achieved executing under the Nios II/e processor, which is 38.75%, is reached executing the 2DLOG algorithm in the "Carphone" sequence with 64 as macroblock size using a window search of 8.

The achieved averages, maximum, and minimum, make the Nios II/e processor the most suitable for achieving constant improvements.

Focusing now on the Nios II/s processor, we can also observe constant results, although they are lower compared to the Nios II/e processor, due to Nios II/s processor

contains an instruction cache which improves the execution of the selected algorithm when not using the custom instruction. Indeed, the achieved time execution saving average is between 40% and 45% in every case when grouping by macroblock size.

Nios II/s processor execution time savings go from 16.22% as the minimum achieved, in the case of executing the 2DLOG algorithm in the "Foreman" sequence with 16 as macroblock size using a window search of 8, to 67.96% as the maximum value, when executing the FST algorithm in the "Foreman" test sequence with 32 as macroblock size using a window search of 32.

These accomplished improvements make the Nios II/s processor not so constant as previously commented for the Nios II/e processor, but good in nearly every case due to the fact that reached execution time savings are around 40%.

Now, regarding the Nios II/f processor, we can see that achieved improvements are generally lower when compared to the two other processors, due to the fact that this processor has both instruction and data cache, which improve the execution of the selected algorithm when not using the custom instruction. We can also observe that, when executing the FST algorithm with macroblock size 64, the improvements are much higher, due to the combination of the data cache with the multi-cycle custom instruction which uses pairs of group of four pixels instead of pairs of single pixels. Indeed, the achieved time execution saving average is around 45% with macroblock size 64. This average is very similar to the reached when executing under the Nios II/s processor. In the case of macroblock sizes 16 and 32, average execution time savings are around 25%, that is to say, lower than the two previous Nios II processors.

Nios II/f processor maximum time saving is 72.60%, which is achieved executing the FST algorithm in the "Foreman" test sequence with 64 as macroblock size using a window search of 32. The lowest improvement under the Nios II/f processor is got when executing the TSST algorithm in the "Foreman" sequence with 16 as macroblock size using a window search of 32, where no time saving is attained. These improvements make the Nios II/f processor the most extreme, since it can achieves both, a null execution time saving and nearly the highest one.

### 6.3.2. Results related to macroblock sizes 16, 32, and 64.

In this subsection, we study the presented results for the test sequences, calculating the average between the results, and focusing on the selected macroblock size. To visualize the achieved improvements when using our custom instruction, Figure 6.16 shows the reached results when turning on our custom instruction for every executed algorithm, every Nios II processor, and every window search size, grouped by the macroblock size selected.

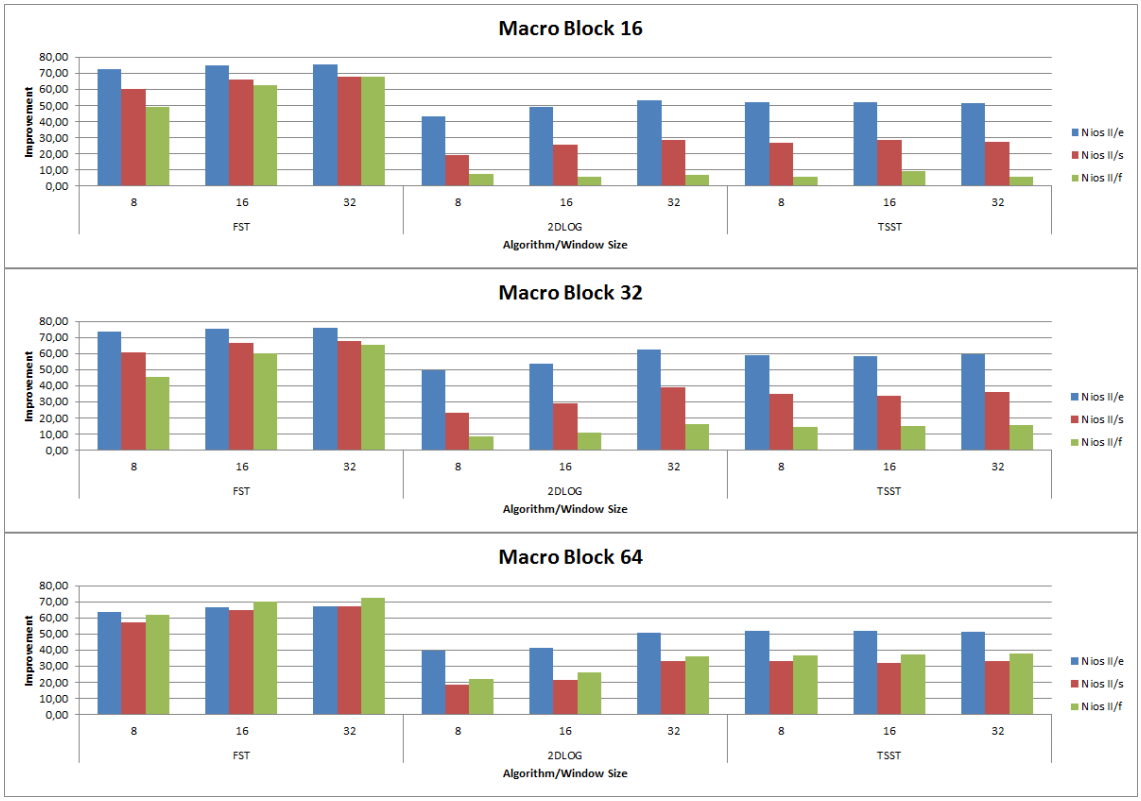


*Figure 6.16: Results focused on macroblock size.*

According to the presented results, using a macroblock size of 16, execution time savings averages go from more than 20%, achieved when executing in the Nios II/f processor, to almost 60%, which is achieved when executing in the Nios II/e processor.

With macroblock size 16, the best execution time saving got is 75.16%, executing under the Nios II/f processor the FST algorithm in the "Foreman" test sequence and using a window search of 8. In the opposite side, we have a null execution time saving, achieved when executing under the Nios II/f processor the TSST algorithm in the "Foreman" sequence using a window search of 32.

These results makes this macroblock size the most inconstant, because achieved results using it go from the minimum achieved in any test (no improvement), to nearly the maximum achieved in any test (75.16%) due to the fact that the custom instruction is called more times.

Now, focusing on the macroblock size of 32, results are more constant. In spite of having execution time savings averages similar to the ones achieved with macroblock size 16 (from more than 25% achieved when executing in the Nios II/s processor to nearly 65% when executing in the Nios II/e processor), its results are more similar to the average.

Using macroblock size 32 the maximum execution time is achieved, 76.08%, obtained executing under the Nios II/e processor the FST algorithm in the "Foreman" test sequence using a window search of 32. It is a bit higher than the maximum achieved with macroblock size 16. The minimum execution time saving achieved is more similar to the average, instead of the one reached when using macroblock size 16. In fact, the minimum throughput got with macroblock size 32 is 7.37%, achieved when executing under the Nios II/f processor the 2DLOG algorithm in the "Carphone" test sequence using a window search of 8. These measures make macroblock size 32 a bit more constant in execution time saving than macroblock size 16.

Regarding macroblock size 64, despite the results are more constant, they are not so high than the other two macroblock sizes, due to the fact that less macroblocks are needed to cover all the frame size, and the custom instruction is called many less times.

With macroblock size 64, the execution time saving averages go from more than 40%, when executing in the Nios II/s processor, to nearly 55%, when executing in the Nios II/e processor. Using this macroblock size, the minimum improvement achieved is a 17.24%, which is achieved executing under the Nios II/s processor the 2DLOG algorithm in the "Carphone" test sequence using a window search of 8. As maximum, this macroblock size achieves a 72.60%, executing under the Nios II/f processor the FST algorithm in the "Carphone" test sequence using a window search of 32. These results make macroblock size 64 the worst for achieving high improvements, but, on the other hand, we remark its constant behavior.

### 6.3.3. Results related to algorithms FST, 2DLOG and TSST.

We deal here with the presented results for the test sequences calculating the average between the achieved results, and focusing on the selected motion estimation technique. To visualize the achieved improvements when using our custom instruction, Figure 6.17 shows reached results when turning on our custom instruction for every macroblock size, every Nios II processor, and every window search size, grouped by the algorithm.

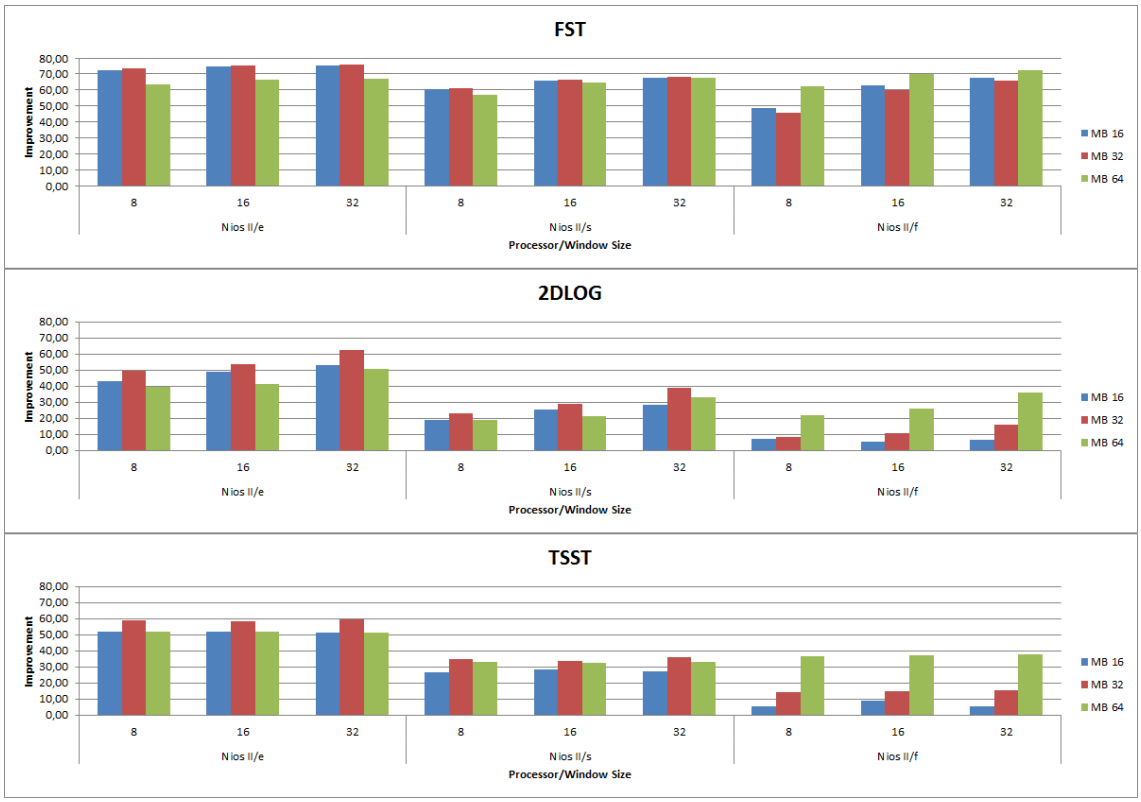


*Figure 6.17: Results focused on algorithm.*

Focusing on the FST algorithm, we can see that the execution time savings achieved are the highest of every algorithm, due to the fact that it is the algorithm with more calls to the custom instruction. Executing the FST algorithm, execution time saving averages are about 65% despite of the macroblock size. Moreover, minimum achieved times savings go from about 45% in the case of executing under the Nios II/f processor using a macroblock size of 32 and a window search of 8, to nearly 60% when executing under the Nios II/s processor using a macroblock size of 64 and a window search of 8.

This fact, suggests that the FST algorithm is the most suitable for achieving higher execution time savings. Indeed, executing this algorithm the highest execution time saving is reached, which is the previous described 76.08%.

Regarding the 2DLOG algorithm, we can conclude that execution time savings are lower than when executing FST due to the fact that it calls many less times the custom instruction.

This is translated in a minimum improvement of 4.63% when executing under the Nios II/f processor the "Foreman" sequence using a macroblock size of 16 and a window search of 16. Apart from this, execution time saving achieved vary in average from more than 25% to a slightly higher than 32% depending on the selected macroblock size.

Focusing on the TSST algorithm, it shows execution time savings similar to the 2DLOG algorithm due to the number of calls to the custom instruction are only slightly lower. The TSST algorithm presents some low minimums like a null execution time saving when executing under the Nios II /f processor the "Foreman" sequence with a macroblock size of 16 and a window search of 32, or a lower than 1% time saving when executing under the Nios II/f processor the same sequence with the same macr block size but using a window search of 8.

In spite of this, execution time savings averages are higher than when executing the 2DLOG algorithm. These averages go from nearly 30% to about 40% depending on the selected macroblock size.

Summarizing these measures, the 2DLOG algorithm is the least suitable for achieving execution time savings. The TSST algorithm is more suitable to achieve good execution time savings although not so good as the FST algorithm. In spite of this, the 2DLOG algorithm achieves a higher maximum, 62.61% when executing under the Nios II/e processor the "Foreman" sequence with a macroblock size of 32 and a window size of 32, than the one achieved executing the TSST algorithm, 60.07% when executing under the Nios II/e processor the "Carphone" sequence with a macroblock size of 32 and a window search of 32.

On the other hand, we have the FST algorithm, whose averages are even more constant than the other two algorithms and also higher, being around 65% despite of the macroblock size. Moreover, its minimums vary from about 45% to nearly 60%, and its maximums go from more than 70% to about 75%, which demonstrates its potential to be improved.

## 6.4. Memory types and memory system designs.

This section firstly presents the memories available in the Altera Cyclone II FPGA which are going to be used in our memory system designs. Secondly, the different memory configuration parameters are explained. Using these memory parameters and combining each one of them, taking into account every possible solution of the selected memories in the FPGA, we will present the achieved results in terms of performance.

### *6.4.1. Selected memory types.*

Although in Section 4.5.3 we have presented every memory type available in our Altera FPGA, we used in the design process only two of them, which were selected in the following manner.

Flash memory was discarded as an option due to its high latency and low capacity. Additionally, non-volatile memory was not needed for this design.

SRAM was discarded additionally because of its cost per MByte, which is more expensive than SDRAM, and designing a low-cost system is of prime importance.

On-chip memory was utilized, since it is the fastest memory type available on the FPGA, and it is also low-cost.

SDRAM was used in the design process too, because of its low cost and very high capacity. Moreover, we needed another memory apart from the one On-chip to allocate all the algorithms and data managed at this work.

### 6.4.2. Configuration parameters in the design process.

Following, the different parameters which will be set to every selected memory type are described. Combining them with every selected memory type, we will have all the presented memory system designs.

- Processor reset vector: it specifies the FPGA memory module where the boot loader code (reset code) is stored, and the reset address, ergo, where the reset vector is located.

- Processor exception vector: it specifies the FPGA memory module where is stored the general exception vector and the address used for it.

- Stack (.stack): it specifies the FPGA memory module where the executed program stack is stored. The stack is mainly used to store the temporary variables and data, and the function call parameters.

- Heap (.heap): it specifies to the FPGA memory module where the executed program heap is stored. The heap is used to store dynamically allocated memory.

- Read/write data (.rwdata): it specifies the FPGA memory module where pointers and read/write global variables are stored.

- Read only data (.rodata): it specifies to the FPGA memory module where read only global variables are stored.

- Program (.text): it specifies the FPGA memory module where the program translated into executable code is stored.

In Figure 6.18, we present an example of how the previously presented sections could be placed, storing all of them inside the same memory module.

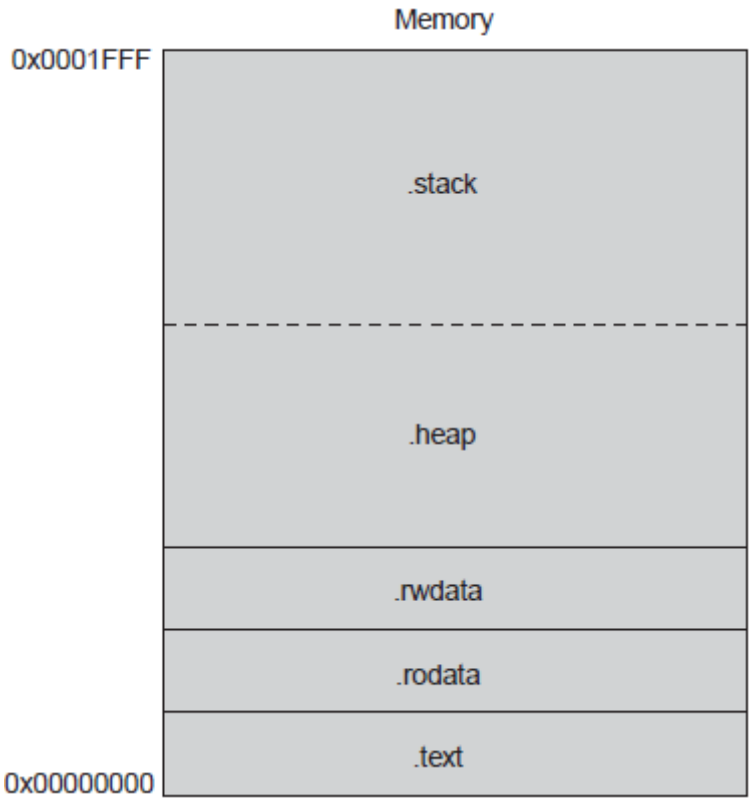


*Figure 6.18: Memory map example [321].*

### 6.4.3. Results using memory system designs.

In Table 6.5, we show the exact memory system designs which obtained a valid design and produces a correct program output, for every combination between the selected memory types and the configuration parameters in our testing platform (Altera FPGA DE2-C35, incorporating a chip Cyclone II EP2C35F672C6N).

Each one of the possible configuration parameters that could be modified by the designer is shown, as well as which kind of the selected memory types was used (On-chip vs SDRAM).

| Design | Processor reset vector | Processor exception vector | Stack (.stack) | Heap (.heap) | Read/write data (.rwdata) | Read only data (.rodata) | Program (.text) |
|---|---|---|---|---|---|---|---|
| 1 | SDRAM | SDRAM | SDRAM | SDRAM | SDRAM | SDRAM | SDRAM |
| 2 | SDRAM | SDRAM | SDRAM | SDRAM | SDRAM | SDRAM | On-chip |
| 3 | SDRAM | SDRAM | SDRAM | SDRAM | SDRAM | On-chip | On-chip |
| 4 | SDRAM | SDRAM | SDRAM | SDRAM | On-chip | SDRAM | SDRAM |
| 5 | SDRAM | SDRAM | SDRAM | SDRAM | On-chip | On-chip | SDRAM |
| 6 | SDRAM | SDRAM | On-chip | SDRAM | SDRAM | SDRAM | SDRAM |
| 7 | SDRAM | SDRAM | On-chip | SDRAM | On-chip | SDRAM | SDRAM |
| 8 | SDRAM | SDRAM | On-chip | SDRAM | On-chip | On-chip | SDRAM |
| 9 | On-chip | On-chip | SDRAM | SDRAM | SDRAM | SDRAM | SDRAM |
| 10 | On-chip | On-chip | SDRAM | SDRAM | SDRAM | On-chip | SDRAM |
| 11 | On-chip | On-chip | SDRAM | SDRAM | On-chip | SDRAM | SDRAM |
| 12 | On-chip | On-chip | SDRAM | SDRAM | On-chip | On-chip | SDRAM |
| 13 | On-chip | On-chip | On-chip | SDRAM | SDRAM | SDRAM | SDRAM |
| 14 | On-chip | On-chip | On-chip | SDRAM | SDRAM | On-chip | SDRAM |
| 15 | On-chip | On-chip | On-chip | SDRAM | On-chip | SDRAM | SDRAM |
| 16 | On-chip | On-chip | On-chip | SDRAM | On-chip | On-chip | SDRAM |

*Table 6.5: Memory system design configuration [320].*

In Figure 6.19, we present three charts resulting from the presented designs in one of our two inputs, the "Foreman" test sequence, for every presented algorithm under the Nios II/e processor. To see the improvement of the memory system design selected, we have fixed window search to 32 pixels and selected macroblock size as 16 pixels, for being the input parameters which make the selected algorithm spend more time, as well as the selected Nios II/e processor.

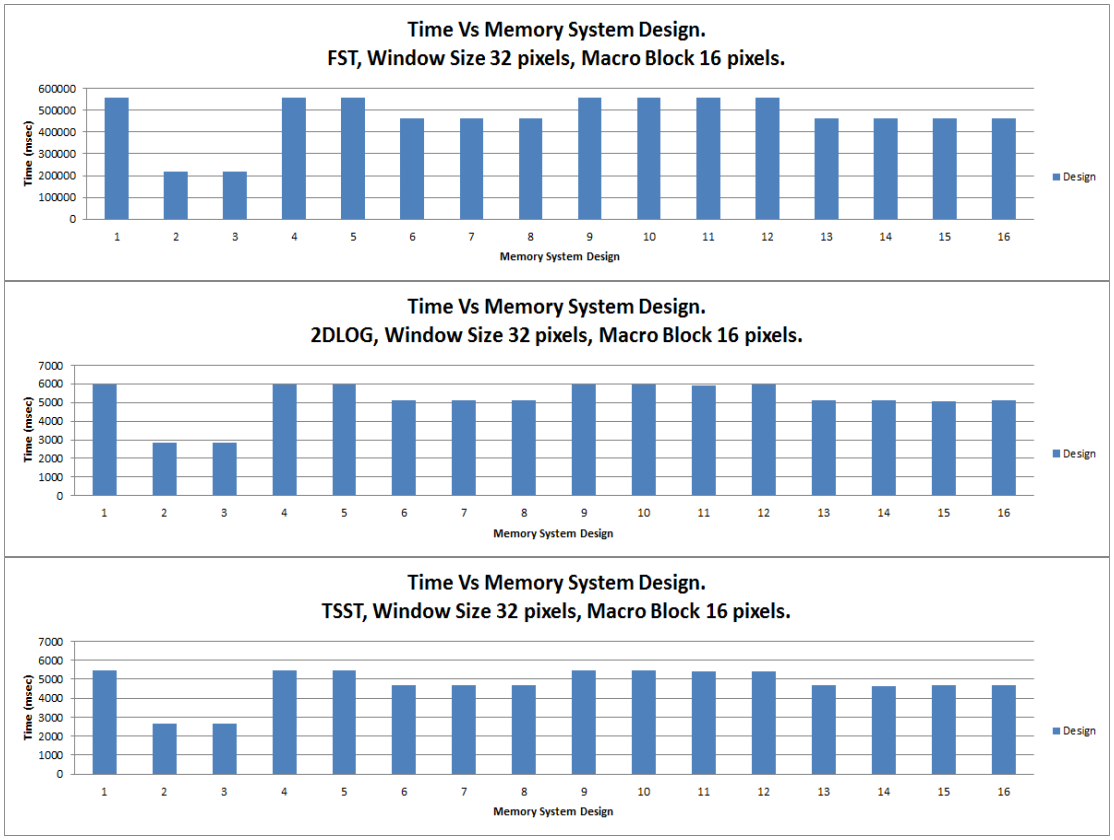


*Figure 6.19: Results for "Foreman" sequence under Nios II/e processor [320].*

As we can see, the group formed by configurations 2 and 3 releases the best performance since the program text is allocated using On-chip memory. The second best group of configurations is formed by designs 6 to 8 and 13 to 16, where the stack is configured to be On-chip. The third group is formed by the remaining configurations (1, 4, 5, and 9 to 12) where stack and program text are allocated both in SDRAM.

The baseline case, design 1, is constructed using SDRAM in every single configuration parameter of the memory system design.

## 6.5. Combinatorial CI combined with memory system design.

Once presented these two previous approaches, -our designed combinatorial custom instruction and the memory system designs-, the embedded system was built by putting them together in order to enhance the performance results.

As demonstrated before, there are only three groups of functional configurations. For this reason, the following results are only presented for configurations 1 to 8 which represent the full range of possible results. Every presented configuration was tested including or not the use of the designed combinatorial custom instruction instead of the GetCost function source code, running the three presented algorithms in every available Nios II processor.

Window search has been fixed to 32 and macroblock size to 16 for being the values that achieves higher execution times spent.

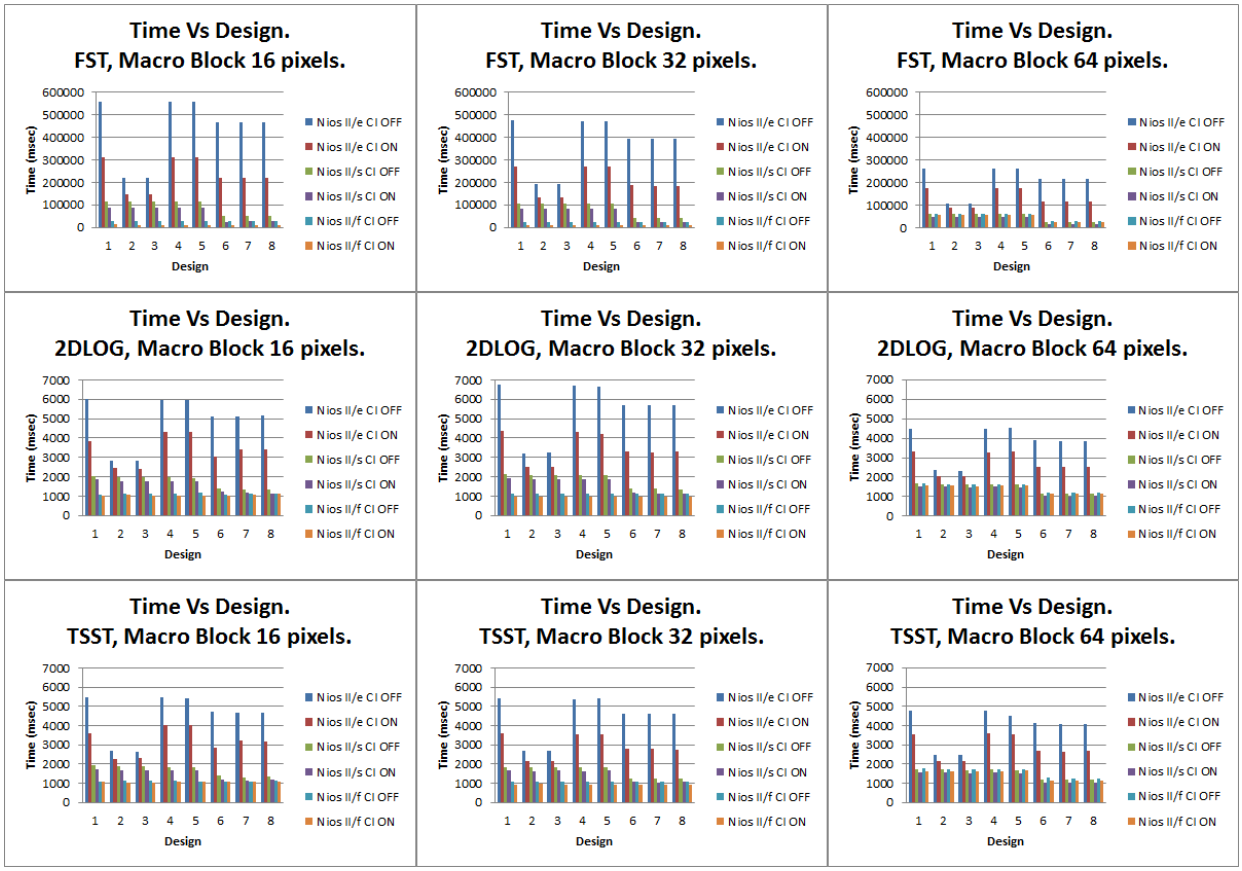


*Figure 6.20: Combinatorial CI + memory design for "Foreman" sequence [320].*

All the described results for the "Foreman" and "Carphone" test sequences are presented at Figures 6.20 and 6.21 respectively.

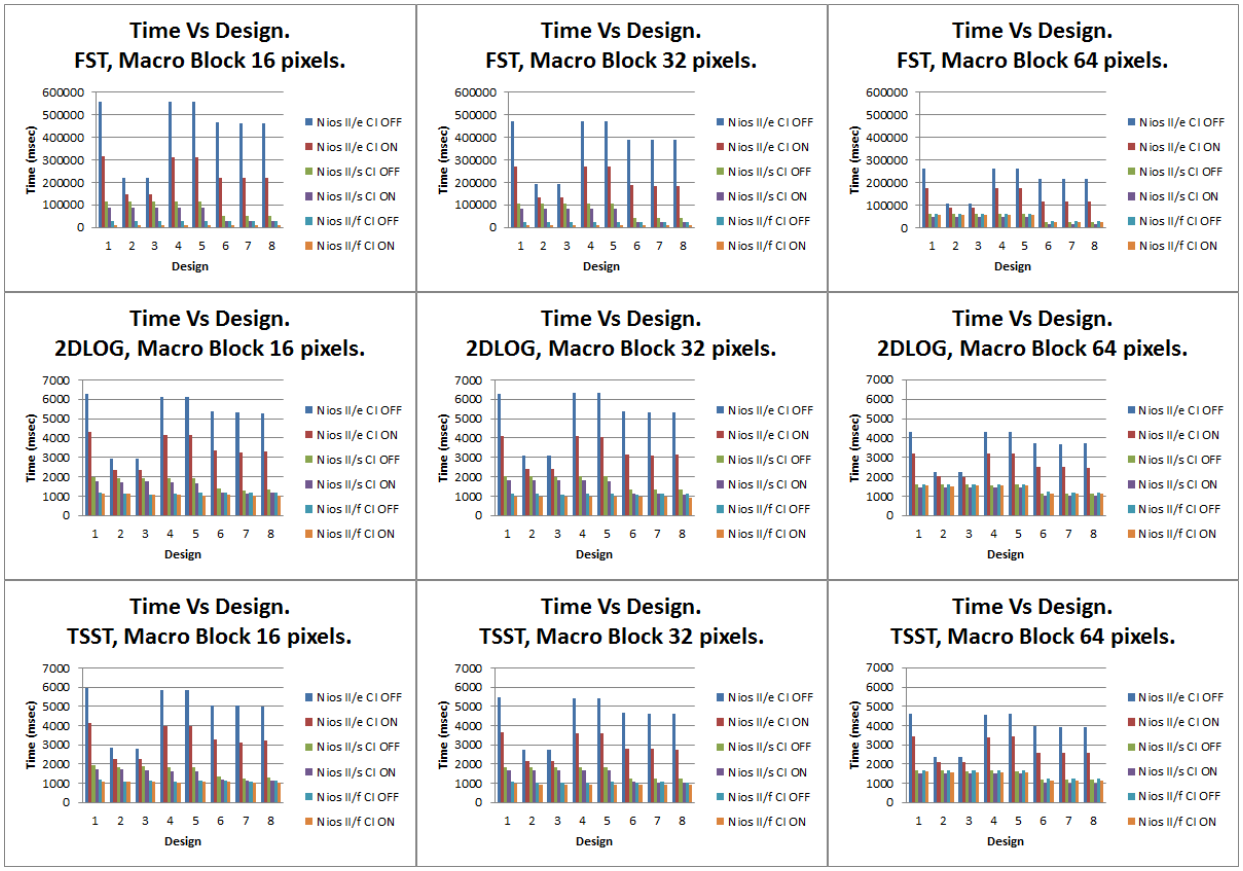


*Figure 6.21: Combinatorial CI + memory design for "Carphone" sequence [320].*

### 6.5.1. Results related to processors Nios II/e, Nios II/s, and Nios II/f.

In this subsection, we discuss the presented results for the two input sequences, but focusing on the selected Nios II processor. In Figure 6.22, the achieved improvements for different memory system designs are presented compared to the reference design number 1 on each case, for every Nios II processor. Color columns represent the improvement when custom instruction is not enabled (CI OFF), nevertheless superposed gray columns show the performance when custom instruction is enabled (CI ON).

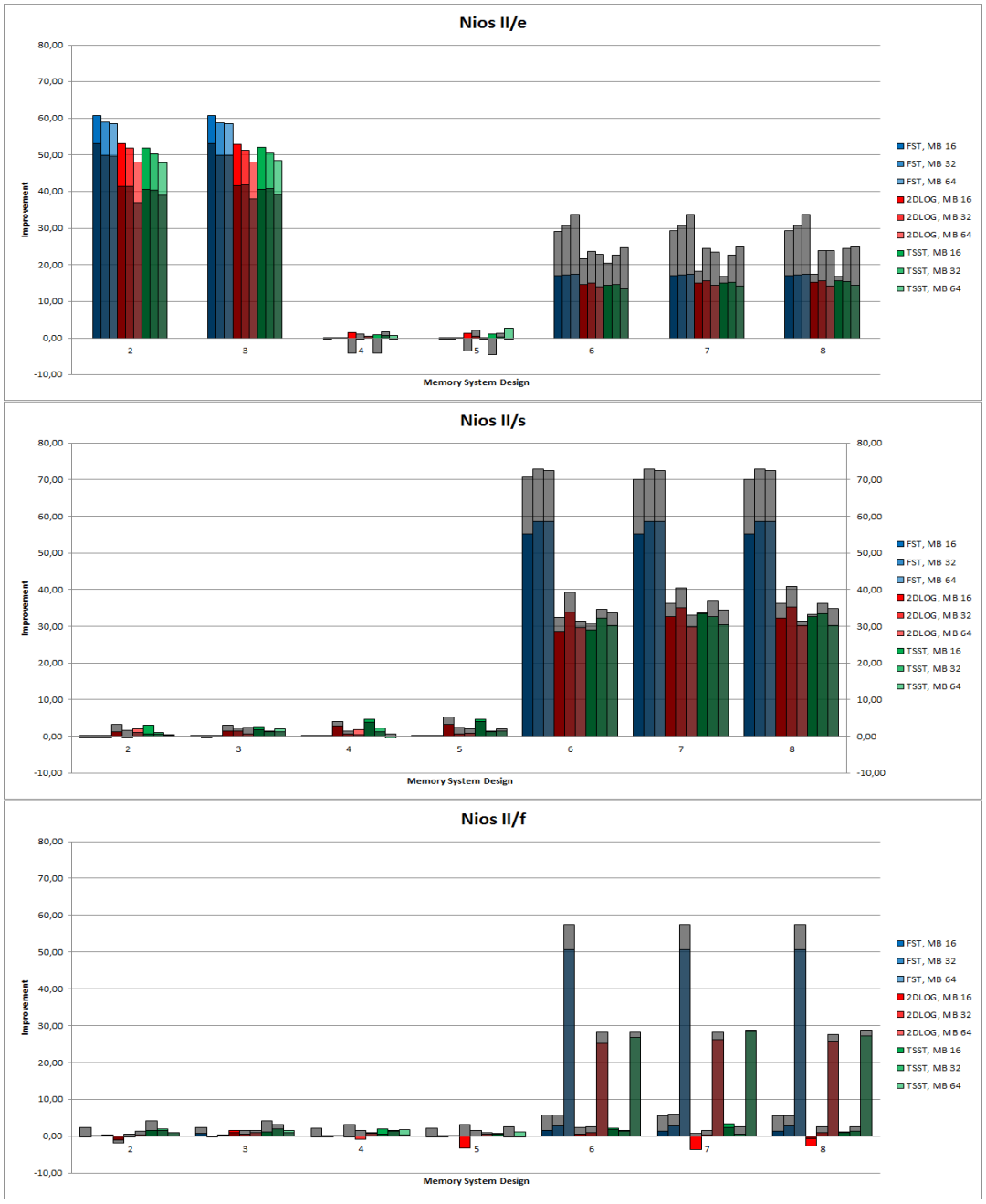


*Figure 6.22: Improvements compared to design 1. Color columns mean CI OFF. Gray columns mean CI ON.*

As we can observe, the achieved results when running the Nios II/e processor are translated into three groups of configurations. The best performance group is formed by configurations 2 and 3 (up to nearly 60%) due to the fact that the program text is allocated in the On-chip memory. The second best performance group is formed by configurations 6 to 8 (up to nearly 35%) due to the fact that the stack is allocated in the On-chip memory. The third performance group is formed by configurations 1, 4, and 5

(up to nearly 3%), due to the fact that both, program text and stack are allocated in the SDRAM memory.

Focusing on the Nios II/s processor, achieved results are translated into two main groups of configurations. The best performance group is formed by configurations 6 to 8 (up to nearly 75%) due to circumstance of stack is allocated in the On-chip memory, and the other one is formed by configurations 1 to 5 (up to nearly 5%) due to the fact that using the instruction cache of this processor allocating program text in the On-chip memory does not have any effect.

Regarding to the Nios II/f processor, presented results are translated into two main groups of configurations. The best performance group is formed by configurations 6 to 8 (up to nearly 60%) due to the fact that the stack is allocated in the On-chip memory, and the other one is formed by configurations 1 to 5 (up to nearly 3%) due to the fact that using the instruction cache of this processor allocating program text in the On-chip memory does not have any effect.

To visualize achieved improvements when using our custom instruction in the different memory system designs, Figure 6.23 show the achieved results when turning our custom instruction on for every executed algorithm in every memory system design, grouped by the selected Nios II processor.

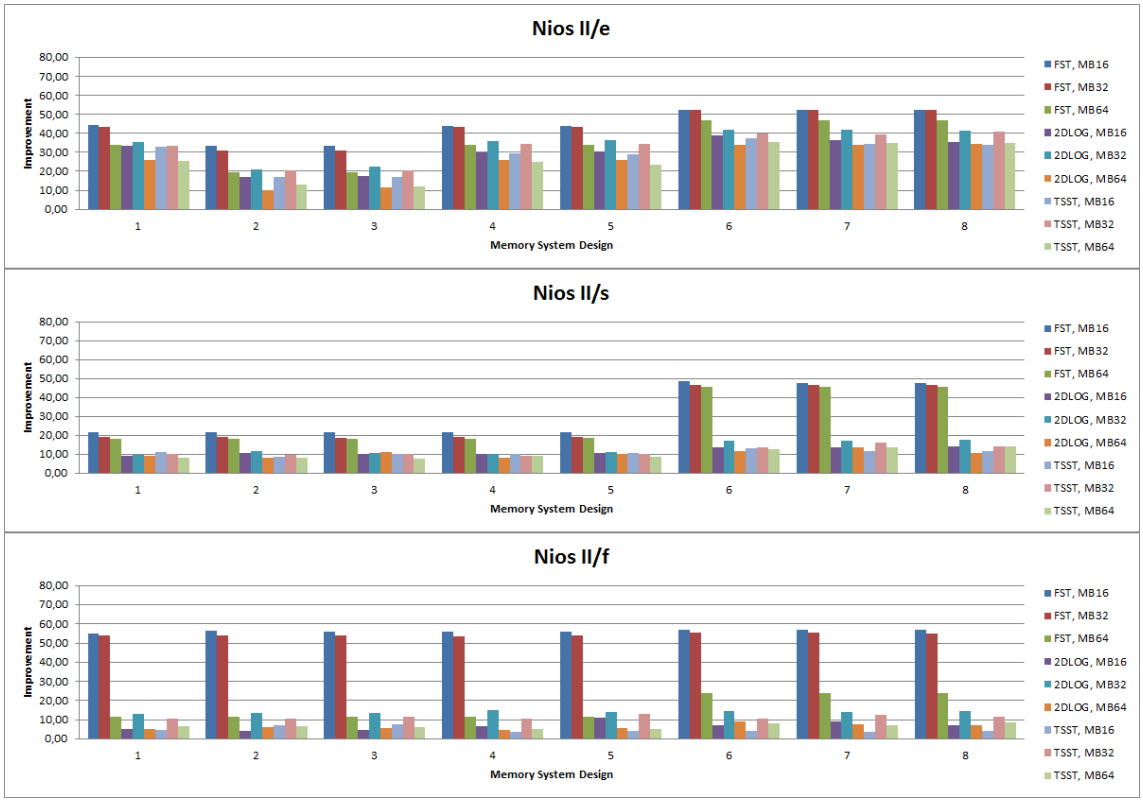


*Figure 6.23: Improvements turning the combinatorial custom instruction on.*

It does not matter the selected macroblock size or the executed algorithm, the Nios II/e processor always achieves better results when activating the custom instruction with the group formed by configurations 6 to 8, between a bit more than 30% to more than 50% depending on the macroblock size and the executed algorithm, due to the stack is allocated in the On-chip memory. In the case of the group formed by configurations 2 and 3, it achieves improvements between more than 10% and more than 30% when turning our custom instruction on, depending also on the macroblock size and the executed technique. Although the group formed by configurations 1, 4, and 5, spends more time executing our combinatorial custom instruction or not, when using it, the achieved improvement compared to the use of the GetCost source code is higher than the one achieved with the group built by configurations 2 and 3. Indeed, it is between more than 20% to more than 40% depending on the referred factors.

In spite of the macroblock size or the executed algorithm, Nios II/s processor always achieves a better performance when turning our custom instruction on the group formed by configurations 6 to 8, due to the fact that the stack is allocated in the On-chip memory. This group achieves improvements from a bit more than 10% to nearly 50%

when turning our custom instruction on, depending on the algorithm executed and the selected macroblock size. On the other hand, we have improvements which vary from nearly 10% to a bit more than 20% depending on those factors, when activating our custom instruction instead of translating the source code into the processor instruction set, due to the fact that using the instruction cache of this processor allocating program text in the On-chip memory does not have any effect.

Regarding the case of turning our custom instruction on, under the Nios II/f processor, all the designs are inside the same group of configurations in spite of the selected macroblock size. This is due to the fact that only the FST algorithm using macroblock sizes 16 and 32 call many more times the custom instruction. Therefore, designs present an improvement from nearly null to more than 50% depending on the executed algorithm and the macroblock size.

Regarding all the Nios II processor types, we can conclude that classification of the memory system designs grouped by the improvements achieved when turning our custom instruction on, corresponds directly with the classification done when grouping the memory system designs by their achieved improvement compared to design number 1, except when we are talking about the Nios II/f processor. In spite of this, the best performance group of memory system designs is not always translated into the best performance group of memory system designs when turning our custom instruction on, as it happens when running in the Nios II/e processor. Although in the Nios II/s processor there is a direct relation between configurations grouped by achieved improvement compared to design number 1 and configurations grouped by achieved improvement when turning our custom instruction on.

### *6.5.2. Results related to macroblock sizes 16, 32, and 64.*

In this subsection, we discuss the presented results for both input sequences, but focusing on the selected macroblock size.

In Figure 6.24, the achieved improvements for different memory system designs are presented compared to our reference, design number 1 on each case, for every used macroblock size.

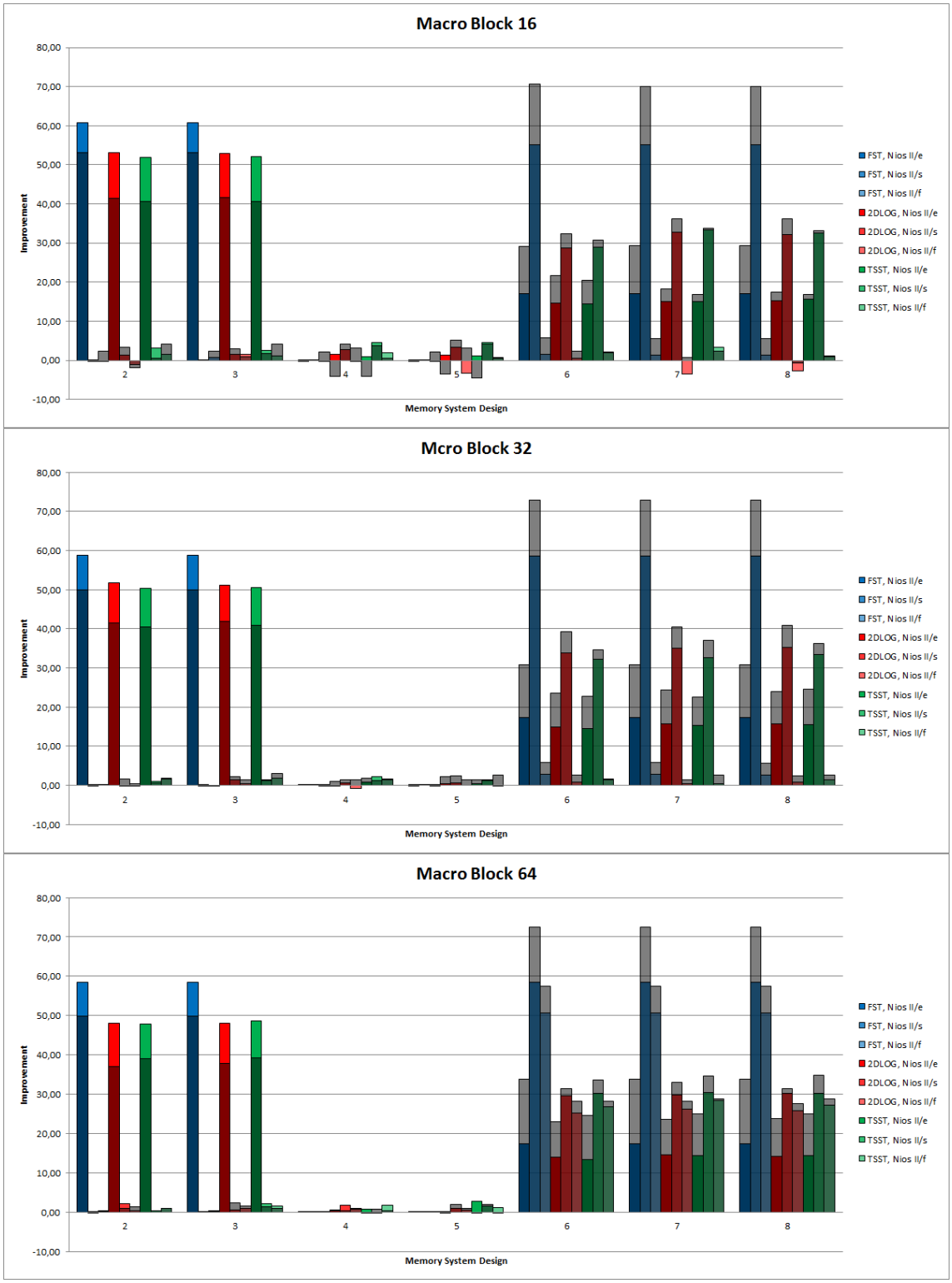


*Figure 6.24: Improvements compared to design 1. Color columns mean CI OFF. Gray columns mean CI ON.*

As we can see, the achieved results are very similar when using macroblock size 16, 32, or 64. Moreover, the mentioned results are translated into three groups of

configurations using the different memory system designs. The best performance group is formed by configurations 6 to 8 (up to nearly 70% using macroblock size 16 and more the 70% using macroblock size 32 or 64) due to the fact that the stack is allocated in the On-chip memory. The second best performance group is formed by configurations 2 and 3 (up to nearly 60%) due to the fact that program text is allocated in the On-chip memory. The third performance group is formed by configurations 1, 4, and 5 (up to nearly 3% using macroblock size 16 and lower using macroblock size 32 and 64) due to the fact that both, program text and stack, are allocated in the SDRAM memory.

To visualize the improvements when using our custom instruction, Figure 6.25 show the results when turning the custom instruction on for every executed algorithm, grouped by the used macroblock size.

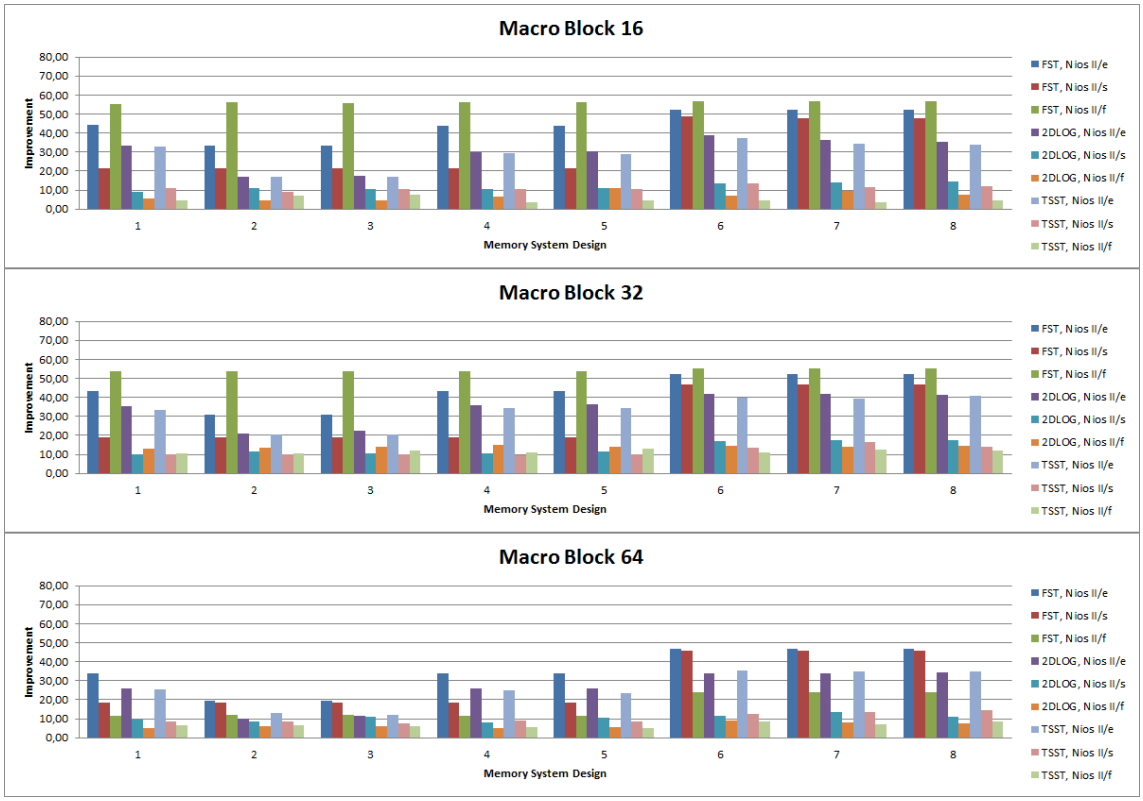


*Figure 6.25: Improvements turning the combinatorial custom instruction on.*

The achieved improvement when turning our custom instruction on using a macroblock size of 16 presents a distribution into only one category, whose improvements vary from nearly 5% to more than 50%.

Regarding the selected macroblock size of 32, it presents the same distribution as macroblock size 16 with the only difference of the minimum improvement from 10%.

Focusing on selected macroblock size 64, we get a distribution divided into three main groups. The best performance group is formed by configurations 6 to 8, achieving improvements from nearly 10% to nearly 50%. The second performance group is formed by configurations 1, 4, and 5, achieving improvements from nearly 5% to more than 30%. And the third one is formed by configurations 2 and 3, achieving improvements from nearly 5% to nearly 20%.

As a conclusion, regarding the distribution presented by memory system designs compared to design number 1 is the same in the three macroblock sizes. Respecting the custom instruction designed, that categorization is the same when selecting macroblock sizes 16 and 32, and different to the distribution presented when selecting macroblock size 64, due to the fact that it is needed many less macroblocks to cover the entire frame and therefore less calls to the custom instruction. So, it does not mind use 16 or 32 macroblock size when selecting a memory system design because of their similar behavior. Additionally, macroblock size 64 release less improvement when turning our combinatorial custom instruction on.

### *6.5.3. Results related to algorithms FST, 2DLOG, and TSST.*

In this subsection, we discuss the presented results for both input sequences together, but focusing on the selected matching algorithm.

In Figure 6.26, the achieved improvements for different memory system designs are presented compared to our reference, design number 1 on each case, for every used motion estimation technique.

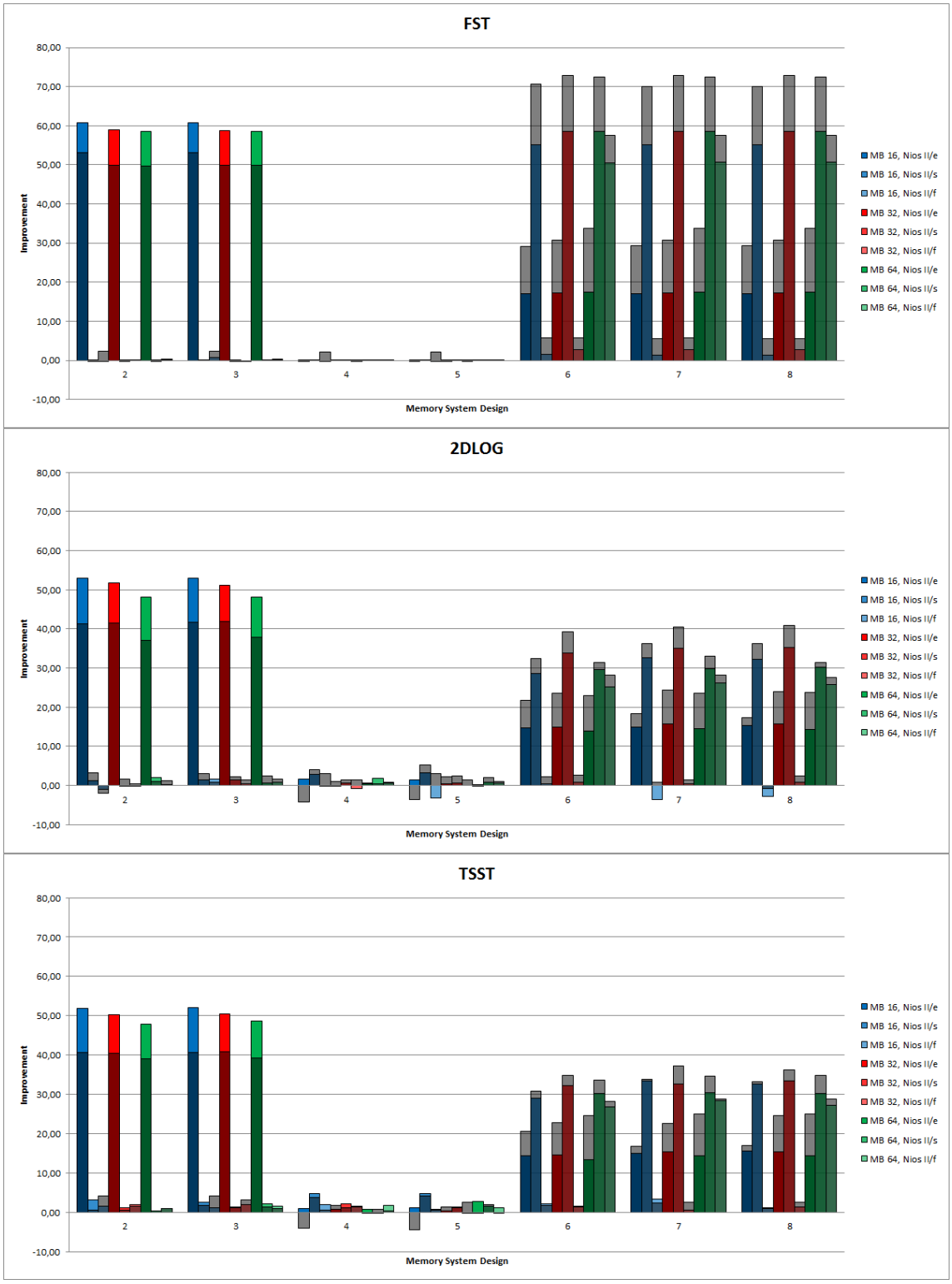


*Figure 6.26: Improvements compared to design 1. Color columns mean CI OFF. Gray columns mean CI ON.*

As we can see, the achieved results when executing any presented technique are translated into three groups of configurations when using the different memory system

designs. The first performance group is formed by configurations 6 to 8 (up to nearly 70% executing the FST algorithm and nearly 40% executing 2DLOG or TSST algorithm) due to the fact that the stack is allocated in the On-chip memory. The second performance group is formed by configurations 2 and 3 (up to nearly 60% executing the FST algorithm and nearly 50% executing 2DLOG or TSST algorithm) due to the fact that the program text is allocated in the On-chip memory. The third performance group is formed by configurations 1, 4, and 5 (up to nearly 2% executing the FST algorithm and a bit more executing 2DLOG or TSST algorithm) due to the fact that both program text and stack are allocated in the SDRAM memory.

To visualize the achieved improvements when using our custom instruction, Figure 6.27 show the achieved results when turning our custom instruction on for every execution grouped by the used algorithm.

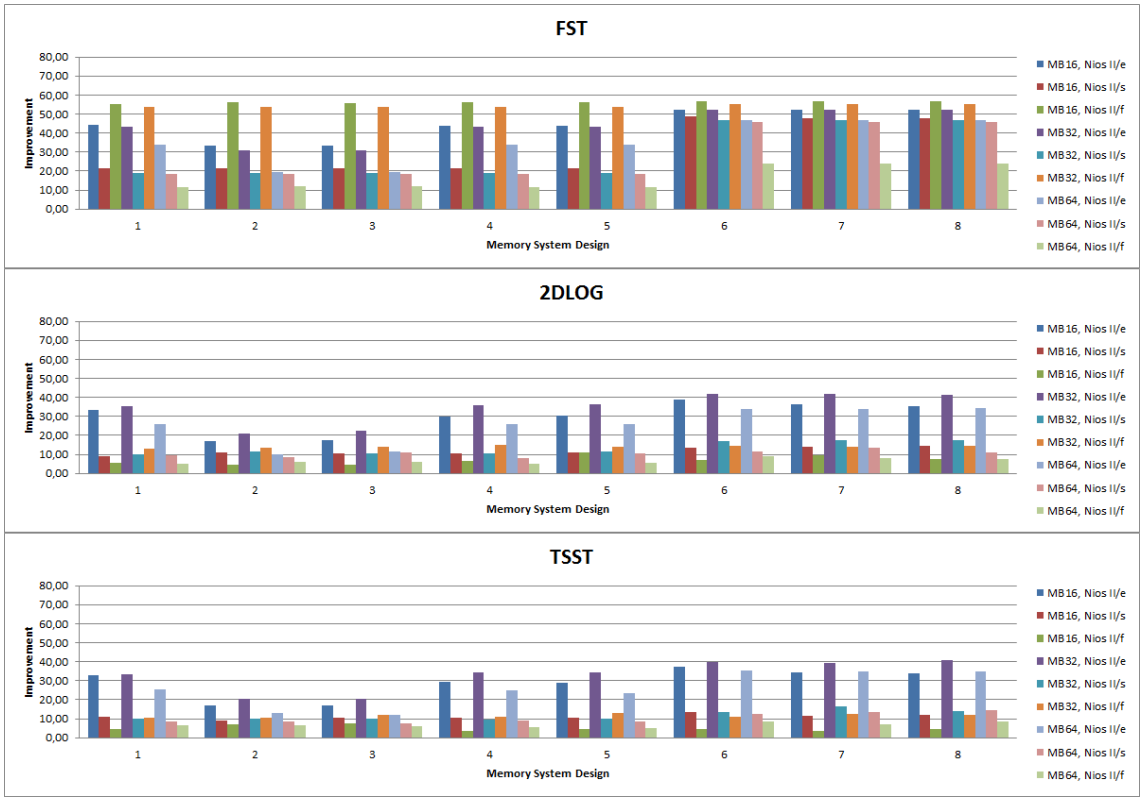


*Figure 6.27: Improvements turning the combinatorial custom instruction on.*

The achieved improvement when turning our custom instruction on executing the FST algorithm presents a distribution divided into two main groups, one formed by

configurations 1 to 5 with improvements from 10% to nearly 60%, and the second one formed by configurations 6 to 8 with improvements from 20% to nearly 60%.

Looking to achieved improvements when turning our custom instruction on executing the 2DLOG algorithm, they present a distribution into three main groups. The best performance one is formed by configurations 6 to 8 (from about 15% to more than 40%), the second one is formed by configurations 1, 4, and 5 (from 5% to about 35%), and the third one is formed by configurations 2 and 3 (from 5% to around 20%).

Regarding the TSST algorithm, when turning our custom instruction on, the improvements are practically the same as the ones presented when executing 2DLOG technique.

For the analysis of Figure 6.25 and Figure 6.26, we can conclude with a similar behavior of 2DLOG and TSST techniques due to the use of input data and the similar number of calls to the custom instruction. On the other hand, FST is the one with the best behavior for getting improvements in both cases due to the high number of iterations, and therefore the frequent access to data and the high number of calls to the custom instruction.

## 6.6. Multi-cycle CI combined with memory system design.

Once presented these two previous approaches, our designed multi-cycle custom instruction and the memory system designs, the embedded system was built by putting them together in order to enhance the performance results.

Again as demonstrated before, there are only three groups of configurations. By this reason, the following results are only presented for configurations 1 to 8 which represent the full range of possible results. Every presented configuration was tested including or not the use of the designed multi-cycle custom instruction instead of the GetCost function source code, running the three presented algorithms in every available Nios II processor.

Window search size has been fixed to 32 pixels and selected macroblock size as 16 pixels, for being the values that achieves higher execution times spent.

In Figure 6.28, all the described results for the "Foreman" sequence are presented at the same time with the results achieved in the base case (no CI), and the results achieved using the combinatorial custom instruction.

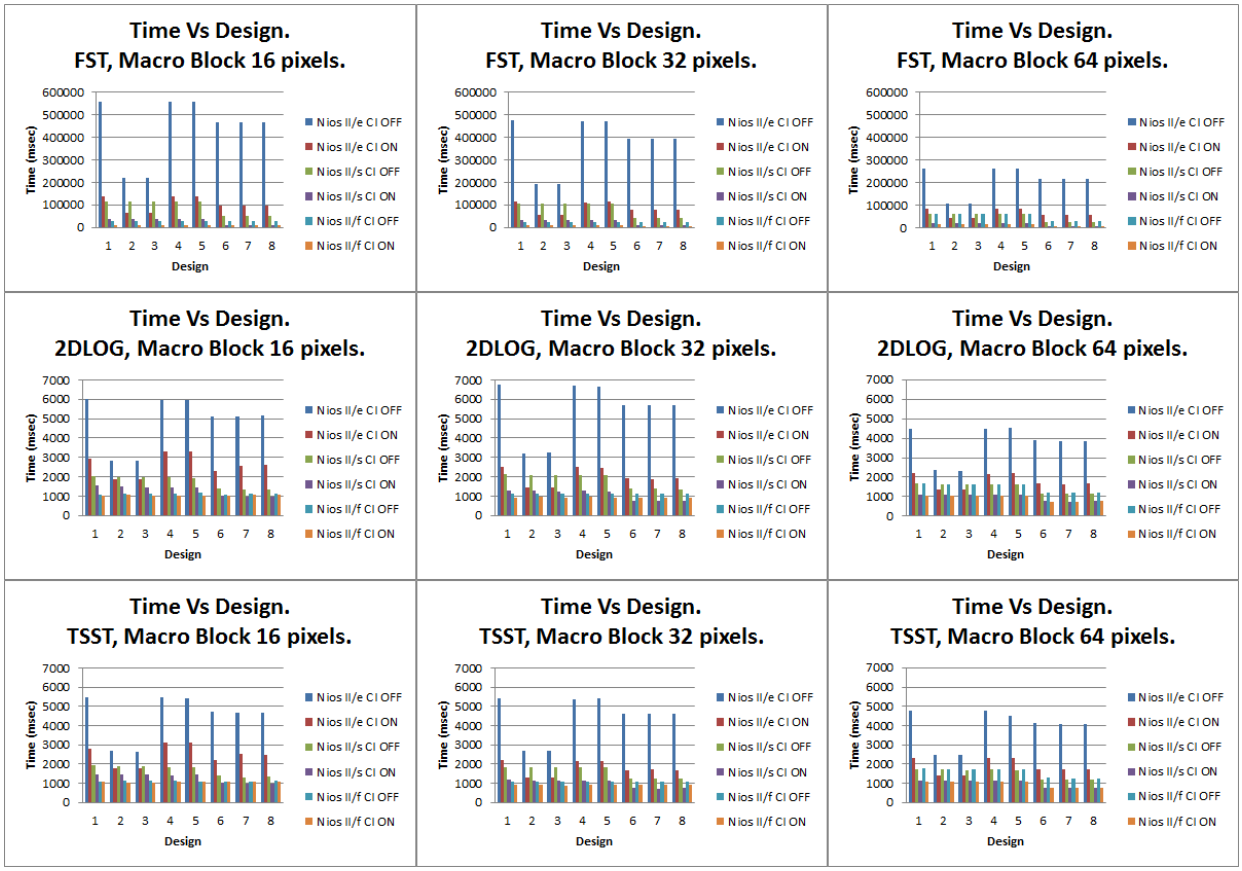


*Figure 6.28: Multi-cycle CI + memory design for "Foreman" sequence.*

Now, Figure 6.29 presents all the described results for the "Carphone" sequence. Figure 6.29 also presents the achieved results in the base case (no CI), and the results achieved using the combinatorial custom instruction to see at a glance the improvement achieves on each case

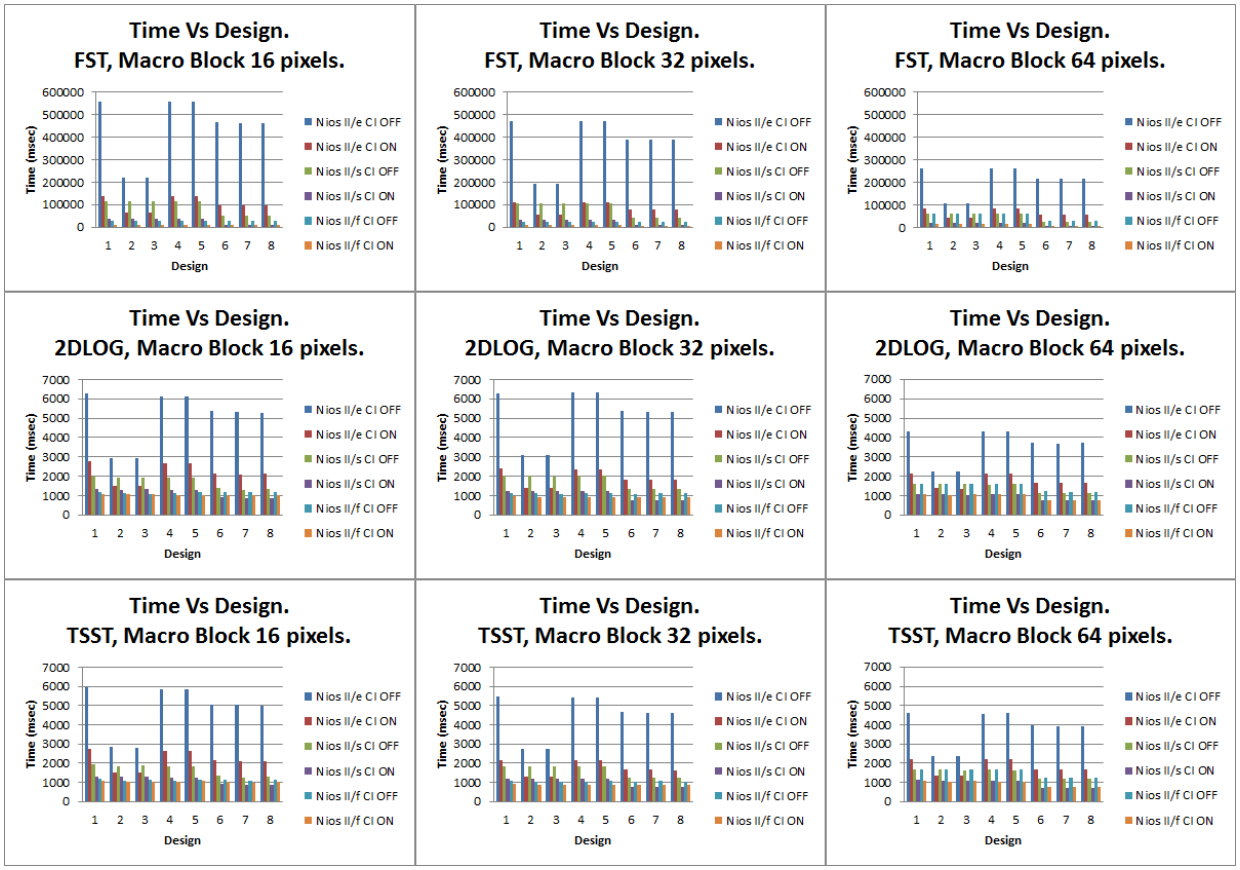


*Figure 6.29: Multi-cycle CI + memory design for "Carphone" sequence.*

### 6.6.1. Results related to processors Nios II/e, Nios II/s, and Nios II/f.

In this subsection, we discuss the presented results when using the multi-cycle custom instruction for the input sequences, but focusing on the selected Nios II processor.

Respecting achieved improvements for different memory system designs compared to the reference design 1, we obtain a similar relative performance than their monocycle counterparts as shown at Figure 6.22. Again we remark to obtain three groups of configurations reaching the best performance by the couple of configurations 2 and 3 (about 60%) for Nios II/ e. For other processors, we find two categories again, receiving the best performance by configurations 6 to 8, up to nearly 75% and 35% respectively for Nios II/s and Nios II/f. In that case all discussion regarding stack allocation is valid here too.

To visualize the accomplished improvements when using the custom instruction, Figure 6.30 shows the achieved results for every executed algorithm grouped by the selected Nios II processor.

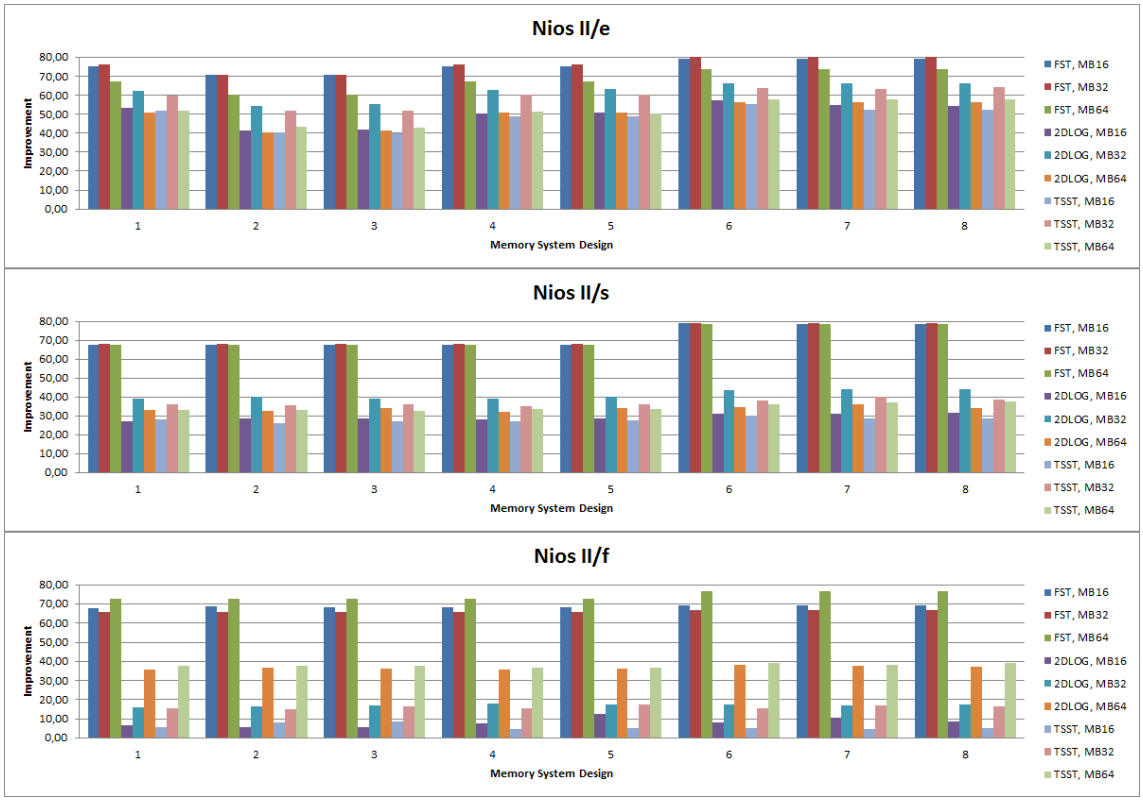


*Figure 6.30: Improvements turning the multi-cycle custom instruction on.*

Following an analogous behavior that the commented in the combinatorial section, it does not matter which macroblock size we select or the executed algorithm: the Nios II/e processor always achieves better results when activating the custom instruction with the group formed by configurations 6 to 8, between nearly 50% to a bit less than 80% depending on the macroblock size and the executed algorithm, due to the fact that the stack is allocated in the On-chip memory. In the case of the group formed by configurations 2 and 3, it reaches improvements between more than 40% and about 70% when turning our custom instruction on, depending also on the macroblock size and the executed technique. Although the group formed by configurations 1, 4, and 5, spends more time independently of executing custom instruction, if this last is activated, the reached improvement is higher than the achieved with the group built by configurations 2 and 3. Indeed, the final performance acceleration increases between 50% to more than 70% depending on the referred factors.

In spite of the macroblock size or the executed algorithm, Nios II/s processor always achieves a better performance when turning our custom instruction on the group formed by configurations 6 to 8 due to the stack is allocated in the On-chip memory. This group achieves configurations from around 30% to nearly 80% when turning our custom instruction on, depending on the algorithm executed and the selected macroblock size. On the other hand, we have improvements which vary from nearly 30% to nearly 70% depending on those factors, when activating our custom instruction instead of translating the source code into the processor instruction set, due to the fact that using the instruction cache of this processor allocating program text in the On-chip memory does not have any effect.

Regarding the case of turning our custom instruction on under the Nios II/f processor, all the designs are inside the same group of configurations in spite of selected macroblock size, although we can observe a slightly higher improvement using macroblock size 64 in design 6 to 8 due to the stack is allocated in the On-chip memory. We can also observe that FST measures obtain higher improvements by far, due to only the FST algorithm using macroblock sizes 16 and 32 calls many more times the custom instruction, and using macroblock size 64 we approach the data cache. Therefore, the designs present an improvement from 5% to around 75% depending on the executed algorithm and the macroblock size.

Regarding all the Nios II processor types, and in similar manner to its custom instruction monocycle counterpart, we can conclude that classification of the memory system designs grouped by the improvements achieved when turning our custom instruction on, corresponds directly with the classification done when grouping the memory system designs by their achieved improvement compared to design number 1, except when we are talking about the Nios II/f processor. In spite of this, the best performance group of memory system designs is not always translated into the best performance group of memory system designs when turning our custom instruction on, as it happens when running in the Nios II/e processor.

### *6.6.2. Results related to macroblock sizes 16, 32, and 64.*

In this subsection, we discuss the presented results for both input sequences but focusing on the selected macroblock size.

Repeatedly, in regard to achieved improvements for different memory system designs compared to the reference design 1, we obtain a similar relative performance than their combinatorial counterparts as shown at Figure 6.24. In like manner, we remark to obtain three groups of configurations, reaching the best performance by the couple of configurations 6 to 8 (up to 70% using macroblock size 16 and more the 70% using macroblock size 32 or 64) due again to the stack is allocated in the On-chip memory. The second best performance group is formed by configurations 2 and 3 (up to nearly 60%) due the program text is allocated in the On-chip memory. Finally, the third performance group is formed by configurations 1, 4, and 5 (up to nearly 3% using macroblock size 16 and lower using macroblock size 32 and 64) having both, program text and stack, allocated in the SDRAM memory.

To visualize the achieved improvements when activating our custom instruction, Figure 6.31 shows the obtained results for every executed algorithm and chosen macroblock size. The performance using a macroblock size of 16 presents a distribution into only one group, whose improvements vary from nearly 5% to nearly 80%.

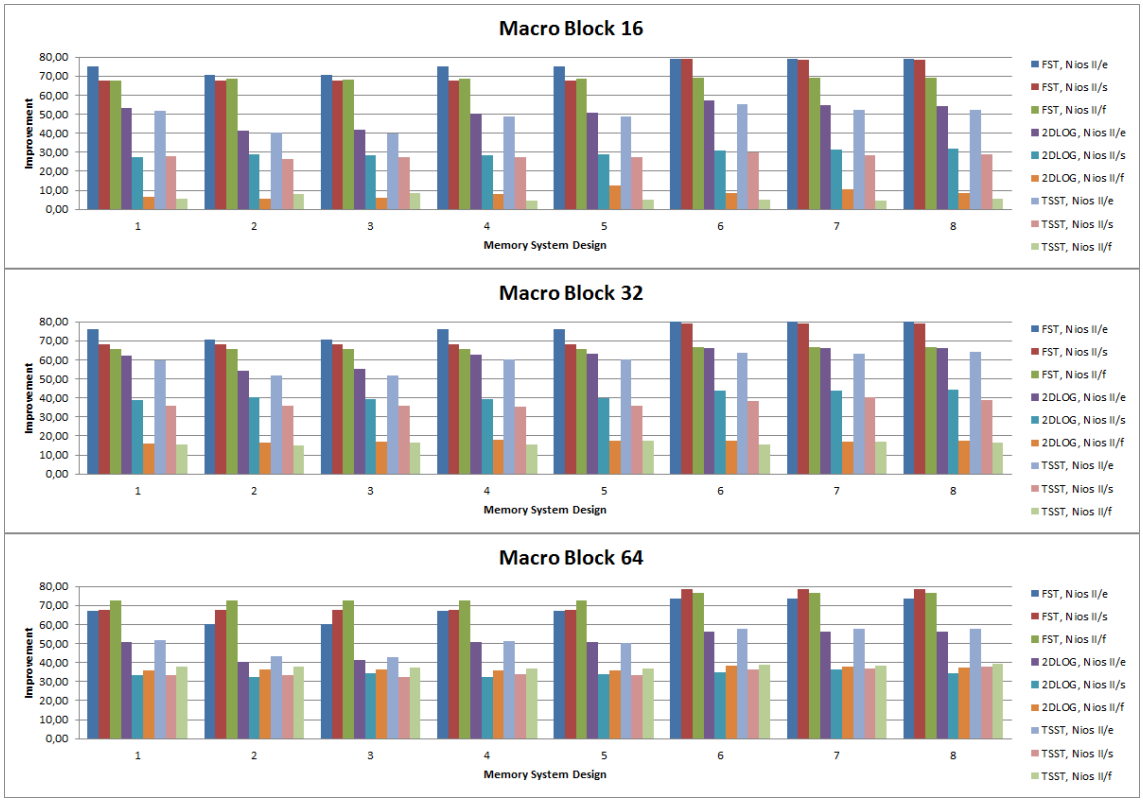


*Figure 6.31: Improvements turning the multi-cycle custom instruction on.*

Regarding the selected macroblock size 32, it presents the same distribution as macroblock 16 is selected. Nevertheless, when macroblock size 32 is selected, obtained improvements are from more than 10% to nearly 80%.

Focusing on selected macroblock size 64, it presents the same distribution as when macroblock size 16 or 32 is selected. But, when macroblock size 64 is selected, achieved improvements when turning our custom instruction on vary from more than 30% to nearly 80%.

As a conclusion, we can observe that the distribution presented by memory system designs is the same in the three macroblock sizes, and the distribution presented when activating our custom instruction is also the same for the three macroblock sizes. So, it does not affect the size used when selecting a memory system design, since the behavior remains constant. Additionally, macroblock size 64 has a better behavior to macroblock sizes 16 and 32 when turning our multi-cycle custom instruction on, achieving to similar maximums but better minimums.

### *6.6.3. Results related to algorithms FST, 2DLOG, and TSST.*

In this subsection, we discuss the presented results for input sequences, but focusing on the selected algorithm.

The performances remains similar than combinatorial cases as show at Figure 6.26. Once more, results when executing any presented technique are translated into three groups of configurations when using the different memory system designs. The first performance group is formed by configurations 6 to 8 (up to nearly 70% executing the FST algorithm and nearly 40% executing 2DLOG or TSST algorithm) due to the stack is allocated in the On-chip memory. The second performance group is formed by configurations 2 and 3 (up to nearly 60% executing the FST algorithm and nearly 50% executing 2DLOG or TSST algorithm) where the program text is allocated in the On-chip memory, and lastly the third one is formed by configurations 1, 4, and 5 (up to nearly 2% executing the FST algorithm and a bit more executing 2DLOG or TSST algorithm) with both, program text and stack, allocated in the SDRAM memory.

To visualize the troughput when using our custom instruction, Figure 6.43 shows the achieved improvements when turning our custom instruction on.

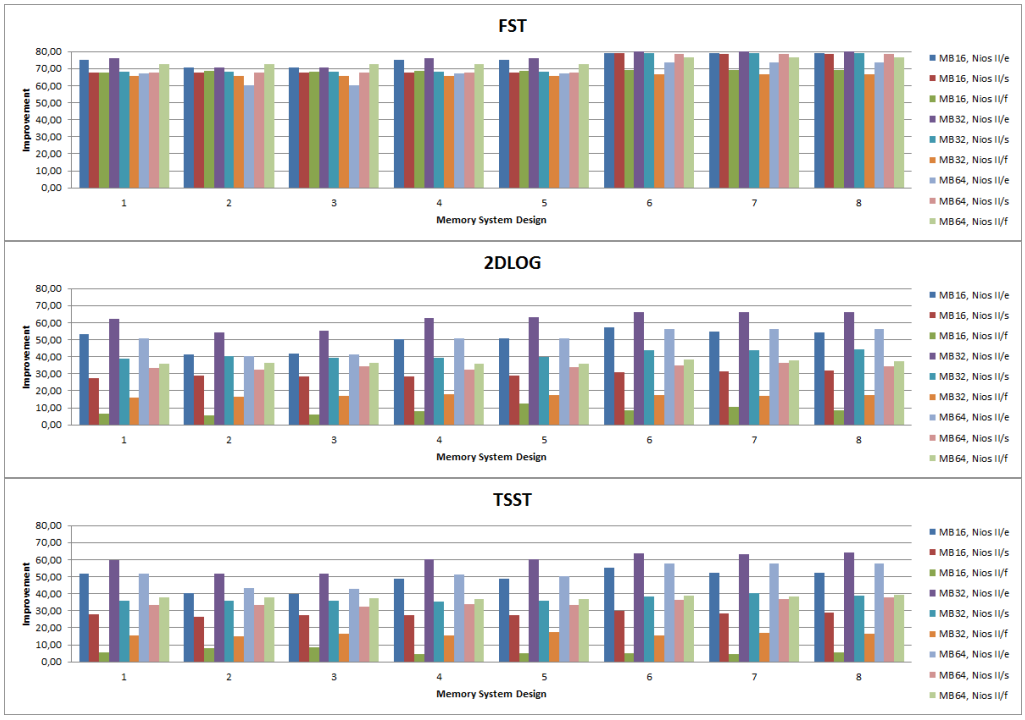


*Figure 6.32: Improvements turning the multi-cycle custom instruction on.*

The achieved improvement when turning our custom instruction on executing the FST algorithm presents a distribution divided into three main groups. The best performance one is formed by configurations 6 to 8 (from nearly 70% to nearly 80%), the second one is formed by configurations 1, 4, and 5 (from nearly 70% to nearly 75%), and the third one is formed by configurations 2 and 3 (from around 60% to around 70%).

Looking to achieved improvements when turning our custom instruction on executing the 2DLOG algorithm, they present also a distribution into three main groups, the same as when executing the FST technique, being the best performance one formed by configurations 6 to 8 (from about 10% to nearly 70%), the second one formed by configurations 1, 4, and 5 (from about 5% to about 60%), and the third one formed by configurations 2 and 3 (from about 10% to around 50%).

Regarding the TSST algorithm, when turning our custom instruction on, the improvements are practically the same as the presented when executing 2DLOG technique.

Also for this multi-cycle analysis, we can conclude that the 2DLOG and TSST algorithms behave similarly due to the use of input data and the number of calls to the custom instruction, although 2DLOG is a bit better for achieving improvements in both cases, -multi-cycle custom instruction and memory system designs-. On the other hand, FST is the one with the best behavior for getting improvements in both cases, due to the high number of iterations, and therefore the frequent access to data and the high number of calls to the custom instruction, without depending on the use of our multi-cycle custom instruction or the selection of a better memory system design than the base case, -design number 1-.

## 6.7. FPGA resources for custom instruction.

In this section, we briefly present the hardware resources[21] that are used in this work, showing at Table 6.6 an overview of the FPGA used resources for each one of the possible FPGA configurations, which provide all the possibilities for implementing all of the tested designs.

Regarding Table 6.6, the FPGA configurations are ordered as follows: the first four rows correspond to the processor Nios II/e, the following four rows correspond to the processor Nios II/s, and the last four rows correspond to the processor Nios II/f. Inside each processor, the group of four rows was categorized in two groups of two rows. The first two rows corresponded to the FPGA configuration without custom instruction, and the last two rows corresponded to the FPGA configuration using the custom instruction. Looking at every pair of rows, it can be seen that the first one contains the chip vectors (the Reset Vector and the Exception Vector) allocated into the On-chip memory, and the second one allocate them into the SDRAM memory. As we can appreciate the hardware resources spent is negligible, concluding the feasibility of constructing either a multi-cycle or combinatorial custom instruction at really low cost.

[21] Compiled and executed with Nios II IDE v10.0 and Quartus II v12.0.

| /Nios II (On-chip or SDRAM) | CI OFF / ON | Logic Cells | Dedicated Logic Registers | I/O Registers | Memory Bits | M4Ks | DSPs | DSP 9×9 / 18×18 | Pins / VPins | LUT-Only LCs | Register-Only LCs | LUT / Register LCs |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| /e (On-chip) | OFF | 2202 | 1050 | 52 | 306176 | 78 | 0 | 0/0 | 56/0 | 1152 | 148 | 902 |
| /e (SDRAM) | OFF | 2198 | 1050 | 52 | 306176 | 78 | 0 | 0/0 | 56/0 | 1148 | 148 | 902 |
| /e (On-chip) | ON | 2352 | 1051 | 52 | 306176 | 78 | 0 | 0/0 | 56/0 | 1301 | 148 | 903 |
| /e (SDRAM) | ON | 2348 | 1051 | 52 | 306176 | 78 | 0 | 0/0 | 56/0 | 1297 | 146 | 905 |
| /s (On-chip) | OFF | 3152 | 1755 | 52 | 341632 | 87 | 4 | 0/2 | 56/0 | 1397 | 228 | 1527 |
| /s (SDRAM) | OFF | 3150 | 1755 | 52 | 341632 | 87 | 4 | 0/2 | 56/0 | 1395 | 225 | 1530 |
| /s (On-chip) | ON | 3284 | 1756 | 5 | 341632 | 87 | 4 | 0/2 | 56/0 | 1528 | 223 | 1533 |
| /s (SDRAM) | ON | 3285 | 1756 | 52 | 341632 | 87 | 4 | 0/2 | 56/0 | 1529 | 223 | 1533 |
| /f (On-chip) | OFF | 3868 | 2175 | 52 | 377088 | 98 | 4 | 0/2 | 56/0 | 1693 | 363 | 1812 |
| /f (SDRAM) | OFF | 3858 | 2175 | 52 | 377088 | 98 | 4 | 0/2 | 56/0 | 1683 | 360 | 1815 |
| /f (On-chip) | ON | 3973 | 2178 | 52 | 377088 | 98 | 4 | 0/2 | 56/0 | 1795 | 360 | 1818 |
| /f (SDRAM) | ON | 3968 | 2178 | 52 | 377088 | 98 | 4 | 0/2 | 56/0 | 1790 | 360 | 1818 |

*Table 6.6: Used hardware resources.*

## 6.8. Final performance results and conclusions.

This work thus, outlines a low-cost system, mapped using very large scale integration technology, which accelerates software algorithms by converting them into custom hardware logic blocks and showing the best combination between On-chip memory and SDRAM for the Nios II processor. The technique developed here has been separately evaluated using a custom instruction paradigm through a combinational instruction and the efficient combination of On-chip memory and SDRAM regarding the reset vector, exception vector, stack, heap, read/write data (.rwdata), read only data (.rodata), and program text (.text) in the design. A combination of two methods was then developed to build the final embedded system.

With the use of mono cycle custom instructions, an improvement was reached of 23%, on average, but close to a 55% improvement in the best case, (Nios II/f , window size of 32, macroblock size of 16 with FST) which supposes a great amount of savings in time spent for execution. With a better use of the memory types available in the design, an improvement of 61% was achieved in the execution time. Putting together both techniques, an improvement of 75% was achieved against the base case.

Regarding multi-cycle custom instruction, the average performance got is about 45% for the full set of parameters: window and macroblock sizes, algorithms and processor architecture used. The maximum throughput using this design is an improvement about 75% (window and macroblock sizes of 32, FST, Nios II/e processor). With the optimization of using the memory types available in the design, an improvement of 60% was achieved in the execution time. Considering, finally the combination of both techniques, an improvement of 80% was reached on average and a 90% for the optimum case.

Following, we are going to present a summary of the work presented on this chapter, comparing the throughput achieved measured in KPPS.

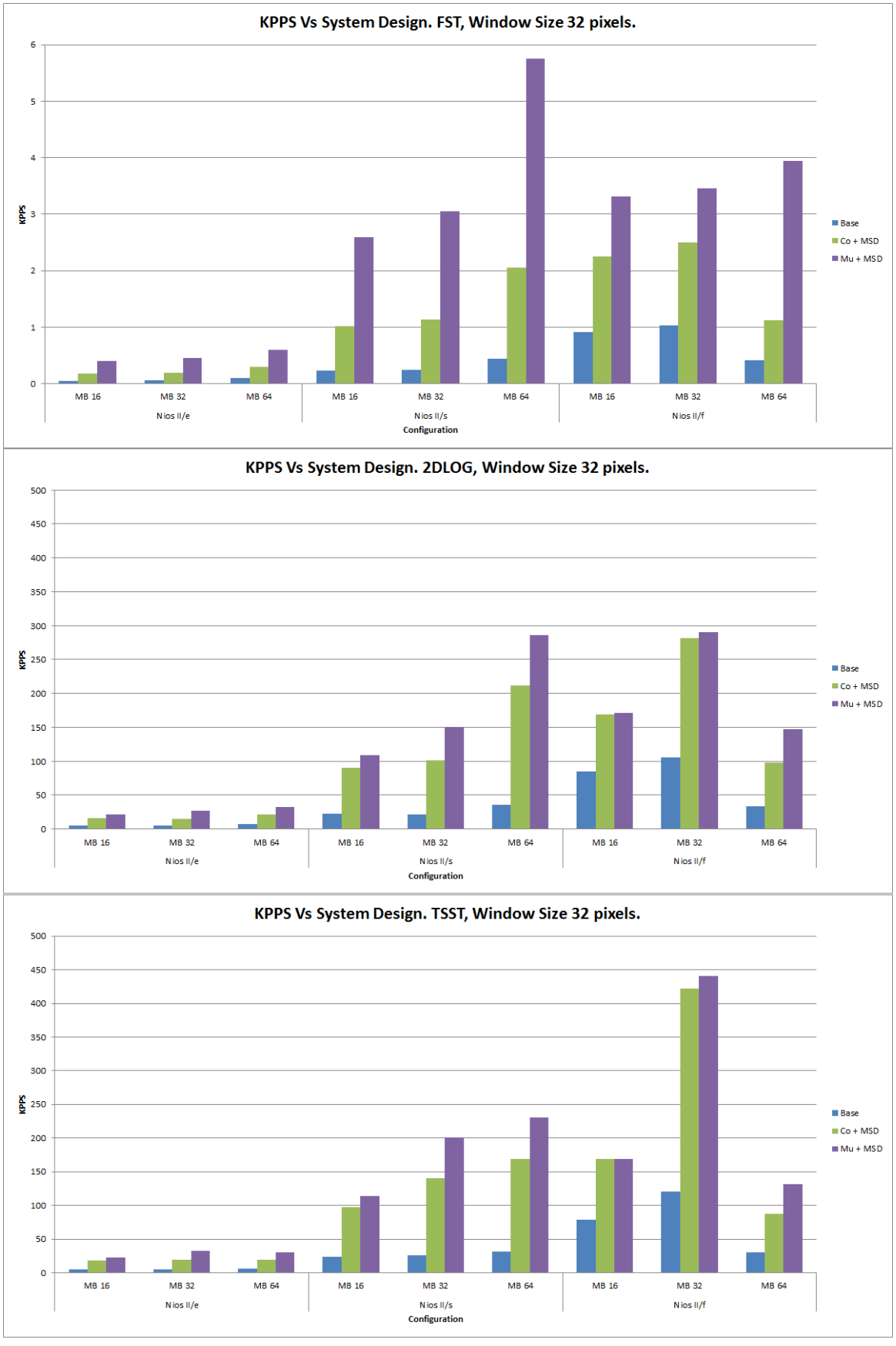


*Figure 6.33: Throughput measured in KPPS for "Foreman" sequence.*

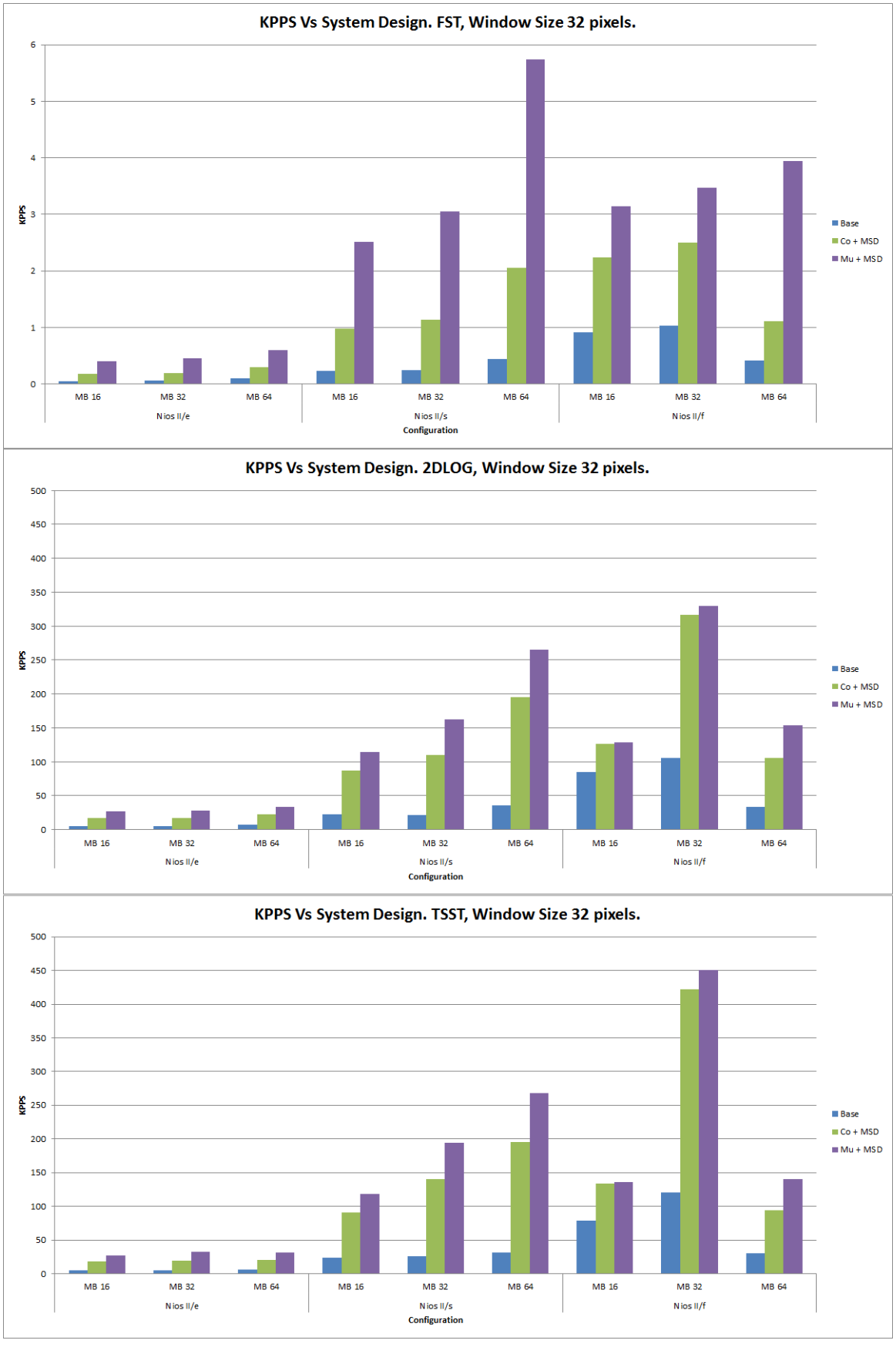


*Figure 6.34: Throughput measured in KPPS for "Carphone" sequence.*

The results achieved have been compared through the base case (no acceleration and every memory parameter set to the SDRAM ), through the use of the combinatorial custom instruction and the best memory system design and through the use of our multi-cycle custom instruction and the best memory system design on each case. Figures 6.33 and 6.34 describe summary the throughput for the "Foreman" and "Carphone" test sequences .

Now, commenting the achieved results, we can observe that looking to the FST technique we almost achieve 6 KPPS in the best case, despite of the used sequence, due to the high number of operations requested by this exhaustive technique with every processed frame. The best case is achieved when using a macroblock size of 64 executing under the Nios II/s processor, due to the exploitation of the instruction cache that this processor provides and the low number of iterations needed when using this macroblock size. Due to all the reasons explained previously along this work, we can accomplish a small sensor of 50×50 @ 2.5 fps.

Focusing on the 2DLOG technique, we achieve nearly 350 KPPS in the best case when executing the "Carphone" sequence, due to the fact that this algorithm needs less number of operations to process a frame than the FST technique. The best case is achieved when using a macroblock size of 32 executing under the Nios II/f processor, despite of the used sequence, due to the use of the instruction cache that this processor provides, and the exploitation of the data cache, that only this processor provides, when using this macroblock size. The final performance obtained suggests that a SoC working with a little sensor of 50×50 @ 130fps would be fully functional.

Regarding the TSST algorithm, we achieve nearly 450 KPPS in the best case when executing the "Carphone" sequence, due to the fact that this algorithm needs even less number of operations to process a frame than the 2DLOG technique. The best case is achieved when using a macroblock size of 32 executing under the Nios II/f processor, despite of the used sequence, due to the use of the instruction cache that this processor provides, and the exploitation of the data cache, that only this processor provides, when using this macroblock size. In this situation we can construct a SoC which processes 50×50 @ 180 fps and 170 fps respectively for either multi-cycle or monocycle approaches.

Despite of the selected macroblock size, the selected Nios II processor, or the sequence used, the best throughput is always achieved using our multi-cycle custom instruction combined with the best memory system design.

# Chapter VII

## Conclusions and future lines

In this chapter we present the main conclusions of this Ph. D. Thesis and our future research.

## 7.1. Conclusions.

Many techniques for accelerating motion compensation routines based on matching approaches have been showed in this research work. Those routines comprise the most expensive and heavy part of many video coding standards, such as H.264, that have been evaluated, analyzed, and profiled.

The first approach presented describes the architecture of a low-cost sensor in an embedded platform, using the Altera C2H compiler in order to accelerate the block-matching motion estimation techniques. This technique is useful for multimedia, image stabilization in robotic and unmanned vehicles, and, recently, for 4-D medical imaging. The Nios II processor allows a plethora of add-ons, such as SDRAM, UART, SRAM, and custom instructions, while embedding everything in a processor by means of an Altera SOPC builder. This approach reduces the peripheral hardware design complexity, enhancing the development of a SoC (System on a Chip). This system has been also characterized in terms of accuracy with the usual PSNR metric for matching systems, resulting in a stable framework that suggests using the FST mode when maximum accuracy is required. At the same time, it is the most hardware consuming configuration, using about 40% for LEs (Logic Element) and embedded DSPs (Digital Signal Processor), and 25% of RAM memory blocks. This system is able to deliver 72.5 KPPS, equivalent to a SoC which processes 50×50 @ 29.5 fps.

The second approach developed in this work addresses the optimization of the algorithms referred previously with the incorporation of custom instructions in the Nios II instruction set, which can contain up to 256 instructions. As previously explained in this memory, a custom instruction is a user defined instruction implemented in hardware inside the processor structure, which performs the operations defined by the user and is added to the Instruction Set Architecture (ISA). Besides, this customized approach is combined with the optimized configuration of combining synchronous dynamic random access memory (SDRAM) with On-chip memory in Nios II processors. A complete profile of the algorithms is completed before the optimization, which locates code leaks, and, afterwards, it creates a custom instruction set, which is then added to the specific design, enhancing the original system.

Additionally, a fully functional optimized memory combination between On-chip memory and SDRAM has been tested to achieve the best performance. The technique developed here has been separately evaluated using a custom instruction paradigm through a combinational instruction and the efficient combination of On-chip memory and SDRAM regarding the reset vector, exception vector, stack, heap, read/write data (.rwdata), read only data (.rodata), and program text (.text) in the design. A combination of two methods was then developed to build the final embedded system.

With the use of combinational custom instruction, an improvement was reached of 23% on average, and about 55% improvement in the best case, which supposes a great amount of savings in execution time. With the optimization of using the memory types available in the design, an improvement of 61% was achieved in the execution time. With the combination of both techniques, an improvement of 70% on average was reached and a 75% for the optimum case against the baseline one.

Regarding multi-cycle custom instructions, we followed the same methodology mentioned, reaching an improvement with respect to the combinatorial approach. The average performance of 44% was reached for the full set of all window search sizes, macroblock sizes, algorithms, and processor architecture used. Additionally, the highest throughput improvement using this design is of about 75% when executing under the Nios II/e processor the FST algorithm using a window size both of 16 and 32, and a macroblock size of 32. This enhancement is kept around 65% if considering all Nios II architectures with the FST technique, for instance, which again is a convenient solution in terms of hardware acceleration. With the optimization of using the memory types available in the design, an improvement of 60% was achieved in the execution time. With the combination of both techniques, an improvement of 80% on average was reached and a 90% for the optimum case against the baseline one.

Regarding the general performance of FST in either combinatorial or multi-cycle design as we remarked previously, we get 6 KPPS in the best case, due to the high number of operations requested with every processed frame by this exhaustive technique. The best case is achieved when using a macroblock size of 64 executing under the Nios II/s processor, because of the exploitation of the instruction cache that this processor provides and the low number of iterations needed when using this

macroblock size. Due to all the reasons explained previously in this work, we can accomplish a small sensor of 50×50 @ 2.5 fps.

Regarding the 2DLOG technique, 350 KPPS are achieved in the best case due to the fact that this algorithm needs less number of operations to process a frame than the FST technique. The best case is achieved when using a macroblock size of 32 executing under the Nios II/f processor, despite of the used sequence, due to the use of the instruction and data cache provided by this processor, when using this macroblock size. The final performance obtained suggests that a SoC working with a little sensor of 50×50 @ 130 fps would be fully functional.

Finally, regarding the TSST algorithm, we achieve nearly 450 KPPS in the best case, due to the fact that this algorithm needs even less number of operations to process a frame than the 2DLOG technique. The best case is achieved when using a macroblock size of 32 executing under the Nios II/f processor, despite of the used sequence, due to the use of the instruction and data cache that this processor provides when using this macroblock size. In this situation we can construct a SoC which processes 50×50 @ 180 fps and 160 fps respectively for either multi-cycle or combinatorial approaches, meaning a real-time motion compensation for the common QCIF format (about 19 fps).

Despite of the selected macroblock size, the selected Nios II processor, or the sequence used, the best throughput is always achieved using our multi-cycle custom instruction combined with the best memory system design. In regard to hardware resources, it can be seen to be slight, either for combinatorial or multi-cycle designs, in contrast with that obtained with C2H compiler approach.

Conclusively, this research work opens the door to motion coding for the low-cost soft-core microprocessors, particularly the RISC architecture of Nios II implemented entirely in the programmable logic and memory blocks of Altera FPGAs. For achieving this, an exhaustive viability study of three very well known matching-based motion compensation algorithms, widely used in multimedia coding, has been performed. Those algorithms have been implemented using on the one hand three different architectural approaches, and on the other hand several algorithmic features. Additionally, with these data set three diverse acceleration techniques have been deployed: C2H, Custom Instruction for combinatorial and multi-cycle design.

Thus, the present work gets together contributions to different research fields like Computer Vision, Multimedia Coding, Block-Matching Techniques, and FPGA based Embedded Systems. Once the consistency of this work has been presented, we can comment three future lines of research that came up in the development of our experiments.

## 7.2. Future goals to achieve.

- Future research lines include plans to integrate a full binocular disparity (stereo matching) method together with the presented motion estimation sensor in an embedded system in order to calculate 3D motion. We plan to extend this system with a larger FPGA than the one used here, and test the whole system in a little robot, autonomous vehicle, or similar structure. In this way, we would have an affordable solution for accelerating matching algorithms while keeping a trade-off between accuracy and efficiency.

- A power consumption analysis stage is also a work that we have planned to do, pairing every custom instruction designed with the performance obtained, for every specific architecture of the microprocessor commented, and again for every matching algorithm and, macroblock and window sizes. Before we are able to do this, we have to tackle the design, implementation and validation of a hardware/software power analysis infrastructure for the low-cost board used in this work (Altera DE2 EP2C35). This infrastructure should be precise, reproducible, and flexible in terms of evaluation of: C2H and Custom Instruction paradigms, other future developments of this work, and finally the impact of techniques and mechanisms for reducing power consumption. A solution involving a trade-off between efficiency and power consumption for low-cost embedded systems based on Nios II processors is expected.

- Additionally, to deal with MPEG-H part 2 also known as H.265 or just High Efficiency Video Coding Standart (HEVC) published in its first version at the beginning of 2013 is also a goal to achieve. For this task, it must be used an FPGA with more resources than the one managed here. Due to the algorithmic complexity involved in this standard, a novel high level methodology such as OpenCL for FPGA is being considered, together with the new compilers CAPS and PGI. Besides, OpenACC, which generates automatically OpenCL code, looks very promising for these intensive

algorithms. It has the advantage of connecting automatically the C descriptive environment supported by specific directives to the silicon domain of logic gates arranged by the FPGAs.

- Regarding another future line of research, that is currently being developed, we will deal with Intellectual Property Protection for embedded systems through lexical obfuscation, a really useful work for the protection of both the multimedia coding algorithm and the content itself. The motivation, methodology, experiments, and results obtained, are deeply explained in the Appendix I of the present memory using on the one hand VHDL and VERILOG, and on the other hand ANSI C for the Nios II processor.

# Appendix I

## Code obfuscation using very long identifiers for FFT motion estimation models in embedded processors

Due to obfuscating the code is most often the only way to protect and avoid reverse engineering, this appendix presents an evaluation of operations widely used in motion estimation for an embedded microprocessor for protection purposes. The implementation of comment methods also allows for the addition of copyright and limited warranty information. The obfuscated code with identifiers of up to 2,048 characters in length is tested for Altera's and Xilinx's field programmable gate arrays for a typical HDL example. Moreover, compiler penalties as well as FFT runtime results are reported.

## 8.1. Introduction.

As we can remarked over this thesis, real-time motion estimation is an important task to be computed using machine vision technology and a multimedia scope with immediate applications and consequences, such as the notable reduction of the bit-rate in video-coding. We have performed an overview of the algorithms and architectures in previous chapters. There are many studies that remark the importance of working in the frequency domain, and using the FFT operation for estimating motion [325 – 331] proposing new approaches. There is a tradeoff solution when estimating motion between (a) ambiguity problem: incorrect estimations due similar objects are located at different places inside a frame and (b) accuracy problem: uncorrected velocity vector estimations due many factors as low spatial resolution of the motion field, noise, shadows, occlusions, illuminations changes., etc. The carefully selection of the parameter in motion estimation will release the ideally goal of low ambiguity and high accuracy, on one hand, taking into account BMMs large-sized image blocks reduce either ambiguity and accuracy due a single large block could include many objects moving towards different directions. On the other hand, short-sized image blocks increase ambiguity due image blocks will contribute similarly in expressions (1) and (2) when their sizes are similar. When properly used, the high frequency components can enhance the accuracy and low components can reduce the ambiguity [326] being the motion information contained in the phase portion of frequency components.

In the scope of matching, the exploitation of FFT using MSE metric, there are many works which employs a novel data structure [332 – 333] independent of image content and not heuristic-based, therefore maximizing the PSNR, identifying the best matching works.

## 8.2. Code obfuscation.

On other hand, digital design strategies are substantially influenced by the shortened time-to-market and even increasing complexity of VLSI circuits. These strategies are based on reusable modules, or intellectual property (IP) cores. Embedded microprocessors (lPs) have become one of the most important IP blocks for FPGA vendors in recent years. Altera for instance reported that they sold 10,000 systems of the

Nios lP development systems in the first 3 years alone [334]. Xilinx reported an even larger download number of their MicroBlaze and PicoBlaze microprocessors based. However, sharing IP cores poses significant security concerns, one of the main concerns being the intellectual property theft of those shared modules. The US Chamber of Commerce estimates that IP theft costs US companies about $250 billion a year, as well as 750,000 jobs. To avoid piracy, legal methods (patents, trademarks, copyrights) and watermarking methods on various levels (system, HDL, or gate level) are necessary.

The VSI Alliance [335] has proposed the use of three approaches for proper protection of IP cores: deterrent approaches, protection approaches, and detection approaches.

Deterrent approaches try to stop attempts for illegal distribution; i.e., using patents, copyrights, and trade secrets. Protection approaches prevent the unauthorized usage of the IP physically by license agreements and encryption. Finally, detection approaches detect and trace both legal and illegal usages of the designs by means of digital signatures, such as digital fingerprinting and digital watermarking, ensuring that a proper course of action can be taken.

Most embedded lP SW protection methods are based on encryption unit placed between microprocessor and program memory such as the execute-only memory proposal, e.g. [336 – 337]. Many different HW protection schemes based on digital watermarking schemes have been proposed in the literature [338 – 341]. Figure 8.1 shows a typical watermarking scheme used in the past. Reverse engineering (i.e., the analysis of the IP to understand the design idea, algorithm, or coding used to solve the task at hand) is not protected by either legal or watermarking methods. When IP is distributed on a source code level, obfuscation is considered a viable choice. Obfuscation is considered a method that transforms source code in a different kind of representation that (1) preserves the function (2) makes reverse engineering difficult, and (3) makes reverse engineering from a business standpoint more expensive than a new code development.

Having enough time and money these transformations can be broken, i.e., reverse engineered. In fact it has been shown by Barak et al. [342] that a complete secure

obfuscation is not possible. The goal of obfuscation is to make reverse engineering more time and cost consuming than to design a new IP block.

In a typical embedded system [343 – 347], a microprocessor core is augmented by co-processors and application software, see Figure 8.2. Obfuscation of the HDL (Verilog or VHDL) and the C source code is required to avoid reverse engineering. Desired IP coding style for reuse has been defined in the OpenMORE program developed by Synopsys and Mentor Graphics and then maintained by the VSI Alliance [348]. High design reuse is confirmed by a high OpenMORE score. A good obfuscated IP block would therefore have a low OpenMORE score. High scores are given for comments, formatting, and identifier choice. These should be the largest targets of an obfuscation tool (as they are used by commercial tools). This report outlines new open source obfuscation tools for VHDL, Verilog and C-programs that allow IP protection (IPP) via obfuscation. The obfuscation cost for the translation of the source code is usually small. The compile-time cost of the obfuscation code should also be reasonable.

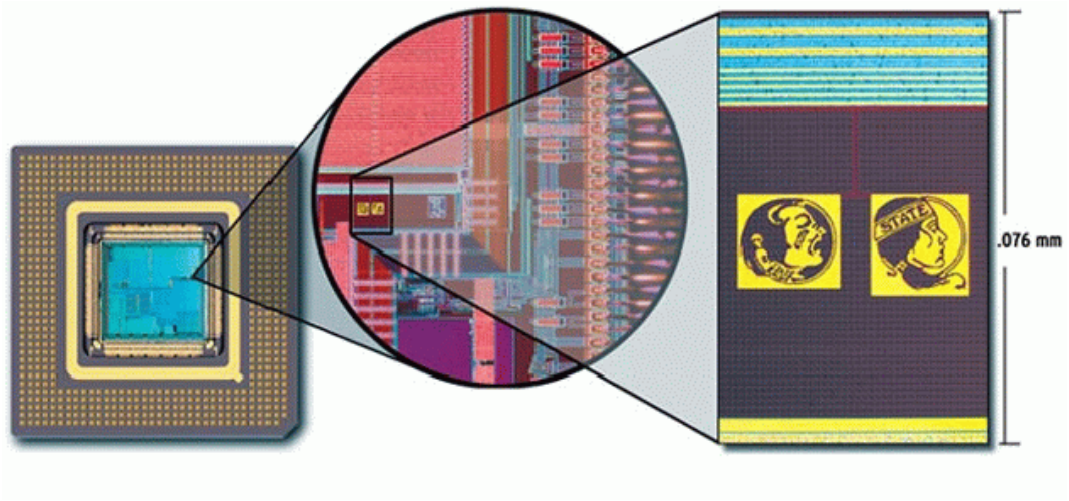


*Figure 8.1: Florida State University logo embedded in layout of the Silicon Graphics MIPS R12000 microprocessor (http://microscopy.fsu.edu/creatures). The microprocessor chip in its ceramic package (center) area on the chip where the logos reside with 9500 magnification, right 92,000 magnification IEEE [338].*

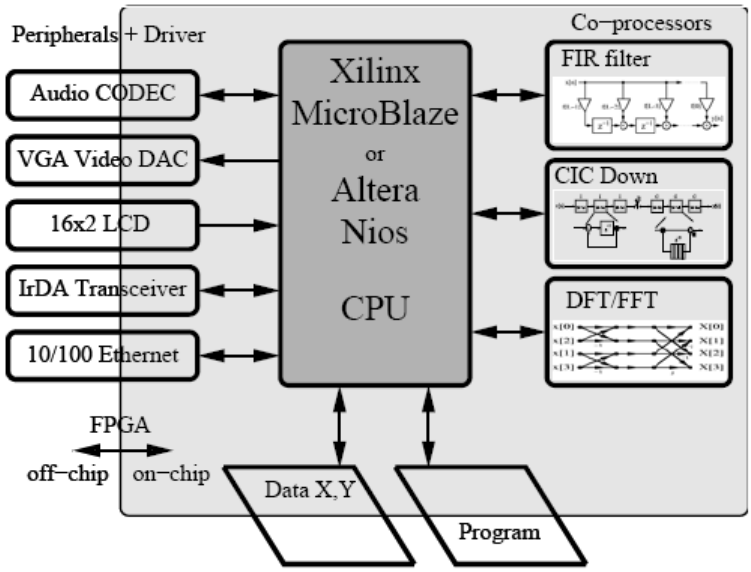


*Figure 8.2: IP core design in FPGAbased embedded microprocessor systems.*

The generated HDL code should not require any additional resources and will run with the same speed. The C-code should maintain the same performance as that of the original code.

For JAVA byte code, many tools are available, since JAVA byte code is particularly vulnerable to reverse engineering. The Open Directory Project (http://www.dmoz.org) currently lists over 20 JAVA obfuscators. Open source (free) tools such as JODE, RetroGuard, and JBCO, as well as commercial tools such as JShrink ($99), JCLOAK ($595), and the second-generation obfuscator Zelix KlassMaster ($399) are readily available. For C-compiler, VHDL, and Verilog the selection is much smaller. Semantic Design, sponsored by an ATP project from NIST, has developed a set of tools for $500 and $1500, respectively. Another first-generation VHDL obfuscator is available from the Spanish company VISENGI that also includes a watermarking capability. A second generation KRYPTON was available from the French company LEDA, but has been discontinued.

## 8.3. Lexical obfuscation method.

Large gain in obfuscation with minimal cost is achieved with lexical transformation and is therefore used as commercial tools by VISENGI or Semantic Design. Collberg et al. [349] gives an overview of possible obfuscation methods.

### *8.3.1. Identifiers obfuscation.*

The identifier names are crucial to the understanding of a design idea. There are different options when obfuscating identifiers.

Short, hard-to-read, or random characters are being used. For example, the code using the '1' and 'l' or the 'O' and '0' identifiers seems to be most annoying, especially when a longer length of the identifier is used in HDL source code such as is then replaced, e.g., by length-16 identifiers as shown in Figure 8.3. Both ANSI C and VHDL have no restrictions for identifier lengths [350 – 351]. The Verilog standard requires a support of at least 1,024 characters for an identifier length [352]. All identifiers in C programs should be replaced. In HDL code the I/O ports should be preserved, so that IP blocks can be used in a larger design, and testbenches can be provided.

```
1  -- VHDL Custom Instruction Template File for
2  -- Internal Register Logic
3  -- pragma(off)
4  -- VHDL obfuscation example (c) 2014 JRTIP
5  -- pragma(on)
6   LIBRARY ieee;
7   USE ieee.std_logic_1164.ALL;  USE ieee.std_logic_arith.ALL;
8   USE ieee.std_logic_signed.ALL;
9   -- ---------------------------------------------------------
10 ENTITY nios_system_mad_0 IS
11 PORT(clk : IN std_logic;
12    ncs_cis0_dataa: IN STD_LOGIC_VECTOR(31 DOWNTO 0);
13                                -- Operand A (always required)
14    ncs_cis0_datab: IN STD_LOGIC_VECTOR(31 DOWNTO 0);
15                                    -- Operand B (optional)
16    ncs_cis0_result: OUT STD_LOGIC_VECTOR(31 DOWNTO 0));
17 END ENTITY;            -- result (always required)
18  -- ---------------------------------------------------------
19 ARCHITECTURE a_custominstruction OF nios_system_mad_0 IS
20  -- local custom instruction signals
21  CONSTANT eight : INTEGER := 8;
22  CONSTANT z4 : STD_LOGIC_VECTOR(3 DOWNTO 0) := (OTHERS => '0');
23  CONSTANT z20 : STD_LOGIC_VECTOR(19 DOWNTO 0) := (OTHERS => '0');
24 BEGIN
25 -- custom instruction logic
26  -- note: external interfaces can be used as well
27    PROCESS(ncs_cis0_dataa, ncs_cis0_datab)
28      VARIABLE a, b, s, d : STD_LOGIC_VECTOR(11 DOWNTO 0);
29    BEGIN
30      -- WAIT UNTIL clk ='1';
31      s := (OTHERS => '0');
32      FOR k IN 0 TO 3 LOOP
33        a := z4 & ncs_cis0_dataa(eight*k+7 DOWNTO eight*k);
34        b := z4 & ncs_cis0_datab(eight*k+7 DOWNTO eight*k);
35        IF a>b THEN -- compute AD
36          d := a - b;
37        ELSE
38          d := b - a;
39        END IF;
40        s := s + d;
41      END LOOP;
42      ncs_cis0_result <= z20 & s;
43    END PROCESS;
44 END ARCHITECTURE a_custominstruction;
```

```
1   -- VHDL obfuscation example (c) 2014 JRTIP
2   library ieee; use ieee.std_logic_1164.ALL; use
3   ieee.std_logic_arith.ALL; use
4  ieee.std_logic_signed.ALL; entity
5  nios_system_mad_0 is port( clk: in std_logic;
6   ncs_cis0_dataa: in std_logic_vector( 31 downto 0)
7  ; ncs_cis0_datab: in std_logic_vector( 31 downto 0
8   ); ncs_cis0_result: out std_logic_vector( 31
9  downto 0)); end entity; architecture
10  a_custominstruction of nios_system_mad_0 is
11  CONSTANT lllllllllllllll: integer := 8; CONSTANT
12  lllllllllllllll: std_logic_vector( 3 downto 0)
13  :=( others => '0'); CONSTANT lllllllllllllll:
14 std_logic_vector( 19 downto 0) :=( others => '0')
15 ; begin process( ncs_cis0_dataa, ncs_cis0_datab)
16  VARIABLE lllllllllllllll, lllllllllllllll,
17  lllllllllllllll, lllllllllllllll:
18  std_logic_vector( 11 downto 0); begin
19  lllllllllllllll :=( others => '0'); for k in 0
20 to 3 loop lllllllllllllll := llllllllllllllll&
21 ncs_cis0_dataa( lllllllllllllll* k +7 downto
22  lllllllllllllll* k); lllllllllllllll :=
23 llllllllllllllll& ncs_cis0_datab(
24  lllllllllllllll* k +7 downto lllllllllllllll* k
25 ); if lllllllllllllll> lllllllllllllll then
26  llllllllllllllll := lllllllllllllll-
27 lllllllllllllll; else lllllllllllllll :=
28  llllllllllllllll- lllllllllllllll; end if;
29  lllllllllllllll := lllllllllllllll+
30  lllllllllllllll; end loop; ncs_cis0_result <=
31  llllllllllllllll& lllllllllllllll; end process;
32  end architecture a_custominstruction;
```

*Figure 8.3: Identifier obfuscation. Original code of a custom mean absolute difference (MAD) co-processor used in a Nios II motion estimation design (up). Length-16 obfuscation (down). Formatting and comments are removed unless enclosed in ‘‘pragma’’ statements (see line 3–5 original code, i.e., obfuscated code line 1); VARIABLEs, SIGNALs and CONSTANTs are replaced. The entity ports are left unchanged.*

### 8.3.2. Comments and formatting.

The comments also carry significant information in the OpenMORE standard, and most (except the I/O ports in HDL) should be removed. However, the problem is that the IP cores usually contain copyright information as well as disclaimer/limited warranty statements included in the HDL code. This is typically put at the beginning or end of the file. VISENGI, for instance, allows a header to be preserved; in VO comments with special syntax are preserved. A more flexible approach is to use a compiler directive to switch off the obfuscation of the comments for critical information. A #pragma off/on switch has been implemented for this purpose in O4C.

In HDL, a comment line starting with – – (VHDL) or // (Verilog) followed by pragma (on/off) allows the encoding to start or stop. All formatting information such as white space, tabs, or indentations should be removed. The line length can be set to a random number, although it shouldn't be so long that typical C and FPGA compiler might crash.

## 8.4. Results.

We have employed three different obfuscators (O4C, O4V, and O4VHDL) with open source executable. For the HDL test a (sqrt.v and sqrt.vhd) design from [353] was used due the intensive use of the square root operation in the PSNR for evaluating the similarity between templates of block pixels, expression 2 aforementioned. The C code obfuscation has been tested with a radix-2 FFT algorithm widely used in the models addressed in Sect. 2.2, running on an embedded Nios II/f processor (50 MHz; 4 KB Data and instruction cache; 8 MB external RAM, with some typical peripherals: JTAG, LEDs, switches and a system timer using 4647 LEs, 4 multiplier 9×9 and total 115,456 on chip bits). The source code comparison with and without obfuscation is shown in Table 8.1. The code size is increased by a factor of 51–236 when the identifier length is increased from the original length to 2,048 in that order).

| | **Identifier length** | **orig.** | **8** | **64** | **128** | **256** | **512** | **2048** | **Factor chance** |
|---|---|---|---|---|---|---|---|---|---|
| VHDL | Size (bytes) | 4046 | 4029 | 9896 | 16683 | 26016 | 57387 | 220438 | 54.5 |
| VHDL | Chars | 1922 | 2536 | 8474 | 15261 | 24761 | 55965 | 218780 | 113.8 |
| Verilog | Size (bytes) | 3695 | 3440 | 8051 | 13940 | 25716 | 49063 | 190579 | 51.6 |
| Verilog | Chars | 1594 | 2055 | 6902 | 12791 | 24567 | 48120 | 189430 | 118.8 |
| GCC | Size (bytes) | 5933 | 6271 | 23557 | 44022 | 82501 | 161142 | 632645 | 106.6 |
| GCC | Chars | 2665 | 4201 | 21393 | 41316 | 80337 | 158436 | 630481 | 236.6 |

*Table 8.1: VHDL, Verilog and C-code original (org.) and obfuscated files sizes.*

### 8.4.1. Altera obfuscation results.

For Altera synthesis results, an EP2C35F672-C4 (having 35 K LEs, 35 multipliers and 105 M4K memory blocks) and a Quartus synthesis tool version 13.0 were used. Table 8.2 reports the Quartus 13.0 synthesis results: performance in size and speed for both HDLs is unchanged, with a small increase in compile-time (<7 %).

| | **Identifier length** | **orig.** | **8** | **64** | **128** | **256** | **2048** | **Max Dev. %** |
|---|---|---|---|---|---|---|---|---|
| VHDL | Compile Time (s) | 17 | 16 | 16 | 16 | 16 | 17 | 5.88 |
| VHDL | LEs | 274 | 274 | 274 | 274 | 274 | 274 | 0.00 |
| VHDL | Mult 9×9 | 2 | 2 | 2 | 2 | 2 | 2 | 0.00 |
| VHDL | Fmax (MHz) | 84.47 | 84.47 | 84.47 | 84.47 | 84.47 | 84.47 | 0.00 |
| Verilog | Compile Time (s) | 15 | 15 | 15 | 16 | 15 | 16 | 6.67 |
| Verilog | LEs | 261 | 261 | 261 | 261 | 261 | 261 | 0.00 |
| Verilog | Mult 9×9 | 2 | 2 | 2 | 2 | 2 | 2 | 0.00 |
| Verilog | Fmax (MHz) | 86.94 | 86.94 | 86.94 | 86.94 | 86.94 | 86.94 | 0.00 |

*Table 8.2: Compile and synthesis data of HDL code using Altera's FPGAs and tools.*

### 8.4.2. Xilinx obfuscation results.

For Xilinx, the ISE 14.7 with the fast synthesis option for a XC3S200 was used. The Xilinx synthesis results in Table 8.3 show that the Place and Route simulated annealing approach used by the ISE software produces a wider range in synthesis results. Performance for speed may differ up to 7 %. The ISE 14.7 tool was successful up to an identifier length of 128, using the synthesis option ‘‘-new_parser yes’’ up to a 512 identifier length could be synthesized. Larger identifier length is currently not supported

and ISE reports the following error message: FATAL_ERROR: Utilities: UtilCnameimp.c:457:1.14—maximum name length exceeded.

| | **Identifier length** | **orig.** | **8** | **64** | **128** | **256** | **512** | **Max Dev. %** |
|---|---|---|---|---|---|---|---|---|
| VHDL | Compile Time (s) | 23 | 23 | 22 | 23 | 23 | 23 | 4.35 |
| VHDL | LUTs | 241 | 241 | 241 | 241 | 241 | 241 | 0.00 |
| VHDL | Mult 9×9 | 1 | 1 | 1 | 1 | 1 | 1 | 0.00 |
| VHDL | Fmax (MHz) | 68.61 | 65.02 | 63.89 | 65.84 | 67.78 | 67.99 | 6.88 |
| Verilog | Compile Time (s) | 23 | 22 | 23 | 22 | 22 | 23 | 4.35 |
| Verilog | LUTs | 239 | 239 | 239 | 239 | 239 | 239 | 0.00 |
| Verilog | Mult 9×9 | 1 | 1 | 1 | 1 | 1 | 1 | 0.00 |
| Verilog | Fmax (MHz) | 68.95 | 66.94 | 67.64 | 69.32 | 66.99 | 66.02 | 4.26 |

*Table 8.3: Compile and synthesis data of HDL code using Xilinx's FPGAs and tools.*

### *8.4.3. GCC obfuscation results.*

Finally, Table 8.4 shows the runtime results for the GCC compiler on Nios II/f. The run time was measured for an FFT length in the range of 8–4,096 points. The runtime is normalized via 5N×log2(N)/T (ms) since radix-2 FFT effort is proportional N×log2(N) [354]. Less than a 1 % difference in performance runtime was measured with and without obfuscated code. The compile-time difference was <7 % when compiling all modules, including the peripheral drivers, and was <5 % for the incremental/recompile program with just the radix-2 FFT.

| FFT length | Identifier length | | | | | |
|---|---|---|---|---|---|---|
| | **orig.** | **8** | **64** | **256** | **2048** | **Max Dev. %** |
| 8 | 1081 | 1081 | 1081 | 1081 | 1081 | 0.00 |
| 16 | 1114 | 1114 | 1113 | 1114 | 1113 | 0.09 |
| 32 | 1135 | 1135 | 1135 | 1135 | 1135 | 0.00 |
| 64 | 1150 | 1152 | 1152 | 1152 | 1150 | 0.17 |
| 128 | 1163 | 1163 | 1161 | 1163 | 1163 | 0.17 |
| 256 | 1166 | 1166 | 1168 | 1168 | 1168 | 0.17 |
| 512 | 1135 | 1139 | 1137 | 1137 | 1137 | 0.35 |
| 1024 | 1024 | 1024 | 1024 | 1024 | 1024 | 0.00 |
| 2048 | 1024 | 1024 | 1022 | 1022 | 1022 | 0.20 |
| 4096 | 1037 | 1035 | 1037 | 1035 | 1035 | 0.19 |
| Compile all (s) | 6.25 | 6.66 | 6.34 | 6.39 | 6.66 | 6.50 |
| Compile increment (s) | 5.422 | 5.437 | 5.203 | 5.422 | 5.454 | 4.04 |

*Table 8.4: Compile and runtime [5×N×log2(N)/T (MS)] data for radix-2 FFT on NIOS II/F processor.*

The ODROID –XU+E board [355] is specially designed for monitoring power consumption of the ARM A15, A7 GPU and DRAM individually. Runtime results for the GCC compiler are reported in Tables 8.5, 8.6, and 8.7. Again, less than a 1 % difference in performance runtime was measured with and without obfuscated code. The compile time difference was <1 % when compiling all modules, including the peripheral drivers, and incremental/recompile time again was <1 % for the program with just the

radix-2 FFT. Given the large difference in source code size (see Table 8.1) of the files, the small increase in compile- and runtime are highly satisfactory.

| | **Identifier length** | | | | | |
|---|---|---|---|---|---|---|
| **FFT length** | **orig.** | **8** | **64** | **256** | **2048** | **Max Dev. %** |
| 8 | 14,43 | 14,43 | 14,14 | 14,43 | 13,87 | 0,25 |
| 16 | 17,59 | 17,17 | 17,59 | 17,17 | 16,39 | 0,11 |
| 32 | 21,22 | 20,61 | 20,61 | 21,22 | 20,61 | 0,05 |
| 64 | 24,87 | 24,87 | 24,87 | 24,87 | 24,04 | 0,49 |
| 128 | 30,06 | 27,74 | 28,85 | 28,85 | 27,74 | 0,08 |
| 256 | 32,79 | 32,79 | 32,79 | 32,79 | 31,36 | 0,14 |
| 512 | 36,07 | 34,35 | 36,07 | 36,07 | 34,35 | 0,33 |
| 1024 | 40,07 | 40,07 | 37,97 | 37,97 | 37,97 | 0,19 |
| 2048 | 42,43 | 40,07 | 42,43 | 42,43 | 40,07 | 0,15 |
| 4096 | 45,08 | 45,08 | 45,08 | 45,08 | 42,43 | 0,37 |
| Compile all (s) | 0,59 | 0,58 | 0,59 | 0,59 | 0,61 | 0,01 |
| Compile increment (s) | 0,48 | 0,46 | 0,48 | 0,48 | 0,47 | 0,01 |

*Table 8.5: Compile and run time [5×N× log2(N)/T (MS)] data for radix-2 FFT on multicore platform Odroid-XU+E [352] under one single core ARM A7 processor.*

| FFT length | Identifier length | | | | | |
|---|---|---|---|---|---|---|
| | **orig.** | **8** | **64** | **256** | **2048** | **Max Dev. %** |
| 8 | 8,39 | 8,49 | 8,20 | 8,39 | 8,39 | 0,11 |
| 16 | 10,02 | 10,02 | 10,02 | 10,16 | 10,16 | 0,08 |
| 32 | 12,02 | 12,44 | 12,02 | 12,02 | 12,02 | 0,19 |
| 64 | 14,72 | 14,43 | 14,43 | 14,43 | 14,43 | 0,13 |
| 128 | 16,39 | 16,39 | 16,03 | 16,39 | 16,78 | 0,26 |
| 256 | 18,98 | 19,50 | 18,98 | 19,50 | 19,50 | 0,28 |
| 512 | 22,54 | 22,54 | 21,86 | 21,86 | 21,86 | 0,37 |
| 1024 | 23,27 | 23,27 | 23,27 | 23,27 | 23,27 | 0,00 |
| 2048 | 25,76 | 25,76 | 24,87 | 25,76 | 25,76 | 0,40 |
| 4096 | 28,85 | 27,74 | 27,74 | 27,74 | 28,85 | 0,61 |
| Compile all (s) | 0,59 | 0,58 | 0,59 | 0,59 | 0,61 | 0,01 |
| Compile increment (s) | 0,48 | 0,46 | 0,48 | 0,48 | 0,47 | 0,01 |

*Table 8.6: Compile and runtime [5×N×log2(N)/T (MS)] data for radix-2 FFT on multicore platform Odroid-XU+E [355] under two cores ARM A7 processor.*

| | Identifier length | | | | | |
|---|---|---|---|---|---|---|
| **FFT length** | **orig.** | **8** | **64** | **256** | **2048** | **Max Dev. %** |
| 8 | 7,93 | 7,84 | 7,84 | 7,84 | 7,93 | 0,05 |
| 16 | 9,62 | 9,49 | 9,37 | 9,49 | 9,75 | 0,14 |
| 32 | 11,63 | 11,45 | 11,45 | 11,27 | 11,63 | 0,15 |
| 64 | 13,61 | 13,61 | 13,61 | 13,61 | 13,61 | 0,00 |
| 128 | 16,03 | 15,68 | 15,68 | 15,68 | 15,68 | 0,16 |
| 256 | 18,03 | 18,50 | 18,03 | 18,03 | 18,50 | 0,25 |
| 512 | 21,22 | 20,61 | 21,22 | 20,61 | 21,22 | 0,33 |
| 1024 | 22,54 | 21,86 | 21,86 | 21,22 | 21,86 | 0,47 |
| 2048 | 24,87 | 24,87 | 24,04 | 24,87 | 25,76 | 0,61 |
| 4096 | 27,74 | 26,72 | 26,72 | 26,72 | 26,72 | 0,46 |
| Compile all (s) | 0,59 | 0,58 | 0,59 | 0,59 | 0,61 | 0,01 |
| Compile increment (s) | 0,48 | 0,46 | 0,48 | 0,48 | 0,47 | 0,01 |

*Table 8.7: Compile and run time [5×N×log2(N)/T (MS)] data for radix-2 FFT on multicore platform Odroid-XU+E [355] under four cores ARM A7 processor.*

## 8.5. Conclusion.

The use of obfuscation methods for common operations widely and intensively used in motion estimation for embedded system design was evaluated. Open source C, VHDL, and Verilog tools have been developed [356], tested for Altera and Xilinx tools and ARM devices. Free-of-charge C,VHDL, and Verilog obfuscation tools have been

posted on the Web [356] being possible to evaluate them. With a code size increase of over a factor of 200, the addition cost in compile-time was reasonable (<7 %). Altera tools did not show any synthesis penalty, while for Xilinx devices and tools up to 7 % speed penalty occurred. For the C program, no noticeable runtime difference occurred. Altera synthesis tools and GCC C-compiler using Nios II and ARM multicore platform could compile identifiers of length 2,048, while Xilinx ISE works up to a length of 512 with special synthesis options. Obfuscation is therefore proved as a viable IPP method for multimedia coding and transmission protection, especially given the small penalty and the success of avoiding reverse engineering.

# Appendix II

## Resumen

En este apéndice se resume el trabajo presentado, en la lengua materna del autor.

## 9.1. Introducción.

A lo largo de la evolución de la humanidad, se ha intentado construir máquinas con capacidades cognitivas y de procesamiento similares al ser humano para desarrollar un gran número de tareas especializadas, facilitando la interacción con el mundo físico.

Respecto al ser humano, la visión [62] representa su sentido más importante y primario, donde se extrae información continua sobre el entorno, reconociendo objetos a través del movimiento, sus colores, y formas. Se percibe profundidad y distancia a través de texturas, sombras, profundidad binocular y color; además, mientras se observa el movimiento de todo lo que rodea al observador, se realizan cálculos exactos en tiempo real, muchas veces inconscientemente, se predicen eventos futuros (tiempo hasta un impacto, por ejemplo), y se evitan situaciones potencialmente peligrosas. Sin embargo, la percepción visual tiene un gran coste asociado, en el que en el que casi más del 50% de la información de entrada al cerebro proviene de los ojos, y el 40% del área cortical cerebral se dedica a procesar información visual.

A la hora de organizar y procesar la información visual se necesitan pues, gran cantidad de recursos computacionales, muchas veces no abordables si se demanda tiempo real, careciendo los sistemas existentes más avanzados de características que tiene el hombre.

Para que las máquinas puedan gestionar la información del entorno que nos rodea, preliminarmente se representará este entorno, que se consigue gracias a las cámaras que plasman esta información en imágenes. Posteriormente, nos encontramos con la necesidad de captar la información de este entorno en un intervalo de tiempo mayor que un instante. Este problema se ha resuelto gracias al vídeo, donde una serie de imágenes consecutivas (fotogramas o *frames*) son adquiridas con un pequeño intervalo de tiempo entre ellas.

Una vez hemos tomado la secuencia de imágenes que corresponden al video, el siguiente paso para analizar la información del entorno consiste en su codificación [8], es decir, la transformación de las imágenes visuales en un conjunto de datos fácilmente analizables, almacenable, y por lo tanto, capaz de ser comunicado o transmitido.

Uno de los principales puntos en la codificación del video, y por lo tanto en su transmisión, es la estimación de movimiento, usada para codificar el movimiento de una fracción de una imagen respecto a otra imagen o conjunto de éstas.

Respecto a la aceleración de la estimación de movimiento, existen numerosos trabajos anteriores respecto a diferentes familias y plataformas, ampliamente analizados en esta memoria. Relativa a la plataforma hardware, se puede optar por un lado por el procesamiento paralelo, ya sea paralelizando en un procesador multicore o usando como coprocesador un dispositivo GPU (*Graphics Processing Unit*) [9 – 10].

Por otro lado, y elegido en este trabajo, está la aceleración de los propios algoritmos gracias a mejoras de ejecución, bien usando la combinación de los tipos de memoria disponibles en un dispositivo de hardware, bien por medio de la adición de módulos de hardware complementarias que eviten al procesador desarrollar todos los cálculos necesarios para la ejecución de dichos algoritmos, o bien personalizando el procesador a través de nuevas instrucciones añadidas a su juego de instrucciones para así poder desarrollar en un menor número de ciclos de reloj las operaciones más comunes dentro de la ejecución de los algoritmos a acelerar.

Dentro de los dispositivos hardware disponibles para llevar a cabo esta "personalización" de los algoritmos, hemos escogido una plataforma basada en FPGA (*Field Programmable Gate Array*)[1] debido a su prototipado rápido, reconfigurabilidad, amplio uso en el mundo académico y de investigación, lo que hace más conveniente para el desarrollo de este trabajo y su integración en aplicaciones del mundo real.

## 9.2. Motivación.

Nuestra motivación en este trabajo es acelerar la ejecución de algoritmos de estimación de movimiento, ampliamente utilizados en estándares de codificación de vídeo como el H.264, usando dispositivos de muy bajo coste basados en microprocesadores empotrados (*soft-core*). Gracias a los avances logrados en este trabajo, los diferentes dispositivos de bajo coste pueden ver incrementadas sus funciones en lo que a codificación y gestión de video se refiere.

Para ser capaces de acelerar los algoritmos elegidos dentro de los disponibles en el campo de la estimación de movimiento, hemos usado tres estrategias diferentes combinando adicionalmente dos de ellas:

La primera, es la aceleración de las principales funciones del algoritmo en relación con el tiempo de ejecución a través del compilador Altera C2H (C a hardware)[2], que sirve para la creación de un módulo de hardware externo al microprocesador, aunque trabaja conjuntamente con éste y que representa el funcionamiento de la parte elegida a acelerar del algoritmo, aliviando al microprocesador de su ejecución y por lo tanto reduciendo su carga de trabajo.

La segunda estrategia, es la combinación de los dos tipos principales de memorias disponibles dentro de la FPGA, SDRAM (*Synchronous Dynamic Random Access Memory*) y On-chip (interna), en los diferentes módulos necesarios para la ejecución de los algoritmos como la pila (*stack*) o el montículo (*heap*), entre otros.

La tercera estrategia, que se combina con la segunda estrategia propuesta, se basa en la adición de una nueva instrucción para el repertorio de instrucciones del microprocesador. Esta nueva instrucción diseñada a medida, representa la parte del algoritmo donde hay una mayor pérdida del tiempo de ejecución. Dicha instrucción personalizada, se presenta como una instrucción monociclo en una primera versión y como una instrucción multiciclo en una versión posterior más avanzada.

Además, como paradigma de aceleración, también se presenta en este trabajo la evaluación de los métodos de ofuscación léxica para operaciones comunes en algoritmos de estimación de movimiento para sistemas empotrados. En esta parte, se evalúa esta ofuscación sobre código C, VHDL, Verilog, junto con las herramientas desarrolladas y probadas por Altera, Xilinx, y dispositivos ARM. Se presenta en este trabajo, un método de ofuscación léxico para IPP (*Intelectual Property Protection*) viable para la codificación multimedia y protección de la transmisión, sobre todo teniendo en cuenta por una parte, la pequeña penalización de ejecución y recursos, y por otra parte, la ventaja de evitar la ingeniería inversa.

Adicionalmente a la mejora en el tiempo de ejecución, también es prioritaria la miniaturización progresiva a lo largo de los años en el tamaño de los dispositivos y a la

par, optimizando su consumo y prestaciones. Este segundo punto de vista teniendo en cuenta el gran abanico de dispositivos existentes, ha sido el segundo impulso en nuestra motivación, haciendo que este trabajo se centre en sistemas de bajo coste, y por lo tanto hacia el menor uso posible del número de puertas lógicas utilizadas para diseñar sistemas de hardware utilizados para la codificación de video.

## 9.3. Estimación de movimiento.

Hay muchos algoritmos y arquitecturas para la estimación de movimiento en tiempo real que son usados frecuentemente, existiendo una gran literatura asociada [28 – 31]. Podemos clasificar los modelos de estimación de movimiento en tres categorías:

- Métodos basados en la correlación (*Block-matching*): este tipo de métodos, también conocidos como métodos de coincidencia de patrones, trabajan comparando posiciones de la estructura de la imagen entre imágenes adyacentes en el tiempo para inferir la velocidad del cambio en cada posición. Además, son probablemente los métodos más intuitivos [38].

- Métodos diferenciales o de gradiente: este tipo de métodos se basan en la intensidad de la imagen en el espacio y el tiempo. La velocidad se obtiene como una relación de las medidas anteriores [39 – 40].

- Métodos de energía: estos métodos están representados por una serie de filtros construidos con una respuesta orientada en el espacio-tiempo para trabajar a ciertas velocidades. Las estructuras utilizadas para este tipo de procesamiento son bancos de filtros paralelos, que se activan para una gama particular de valores [41].

## 9.4. Estado del arte para estimación de movimiento.

En la estimación de movimiento, sobre todo en el contexto de este trabajo de investigación, es interesante obtener resultados precisos de manera eficiente en tiempo real, utilizando técnicas que permiten su adaptación a los problemas reales. Para ello, se ha realizado en el Capítulo II una revisión de los trabajos de los últimos años en lo que a aceleradores de estimación de movimiento se refiere, incluidos aquellos realizados en torno a GPUs, microprocesadores embebidos, ASICs (*Application Specific Integrated Circuits*) o FPGAs.

## 9.5. Algoritmos de correspondencia de bloques.

El objetivo de los algoritmos de correspondencia de bloques, consiste en calcular los vectores de movimiento del fotograma actual, comparando cada bloque de dicho fotograma con cada macrobloque dentro de una ventana de búsqueda específica y constante, respecto a una imagen de referencia [246 – 247]. El enfoque más simple para esta técnica es una búsqueda exhaustiva, comparando respecto a todos los macrobloques dentro de una ventana de búsqueda en la imagen de referencia, para así estimar el macrobloque óptimo, minimizando la medida de error o BME (*Block-Matching Error*).

Existen varias definiciones para el BME, aunque las más utilizadas son el SAD (*Sum of Absolute Difference*) y el MSE (*Mean Squared Error*), por cada píxel entre un macrobloque de la imagen actual y un macrobloque de la imagen de referencia. Dadas estas métricas de error, la ingente cantidad de cálculos requeridos para calcular el error por estos algoritmos es demasiado alta, por lo que con el fin de optimizar el cálculo de la estimación de movimiento se han propuesto muchos algoritmos de búsqueda mejorados. Dichos métodos se pueden organizar en dos grandes familias:

- SR (*Search Reduction*): estos algoritmos se basan en reducir el número de bloques a tratar dentro de la ventana de búsqueda [248 – 251] mediante patrones de búsqueda estáticos cuyo objetivo es alcanzar el macrobloque con menos BME. La desventaja de estos esquemas reside en no garantizar el evitar óptimos locales.

- CR (*Calculation Reduction*): se reduce el número de cálculos a efectuar con la idea de que no todos los macrobloques tienen que ser comparados con el actual, así que a través de un algoritmo de selección sólo algunos macrobloques dentro de la ventana de búsqueda son elegidos para ser comparados con el macrobloque actual, descartando de forma rápida vectores de movimiento no válidos [252].

### *9.5.1. SR (Search Reduction).*

A continuación, presentamos los tres principales algoritmos de esta familia usados en este trabajo de investigación:

- ***FST (Full Search Technique).***

Este algoritmo es el más intuitivo, sencillo, y preciso, ya que compara respecto a todos los posibles bloques dentro de la ventana de búsqueda en la imagen de referencia para encontrar el que tenga el mínimo SAD.

- ***2DLOG (Two Dimensional Logarithmic Search).***

Este algoritmo [253] comienza con una zona de búsqueda aproximada, que se va ajustando a medida que el vector estimado actual está en el centro de la zona de búsqueda, en la Figura 9.1 se ilustra un ejemplo de uso.

Esta técnica utiliza un patrón en cruz (+) en cada paso con un tamaño de paso inicial de d/4. El tamaño del paso se divide por la mitad sólo cuando el punto con el mínimo SAD de la etapa anterior es el centro o alcanza el límite de ventana de búsqueda. En otro caso, el tamaño de paso permanece fijo. Cuando el paso se reduce a uno, los ocho puntos de comprobación adyacentes al centro de esta etapa son comprobados. El primer paso (color azul) logra el mejor punto en la parte superior. Después, la etapa dos (color verde) consigue el mejor punto a la derecha tal y como lo hace el tercer paso (color amarillo). El cuarto paso (color rojo) logra el mejor punto en el centro por lo que el tamaño del paso se reduce a la mitad. En la quinta etapa (color marrón) se reduce el tamaño del paso debido a que se alcanza el límite de la ventana de búsqueda. Finalmente (color gris), se toma uno como tamaño del paso.

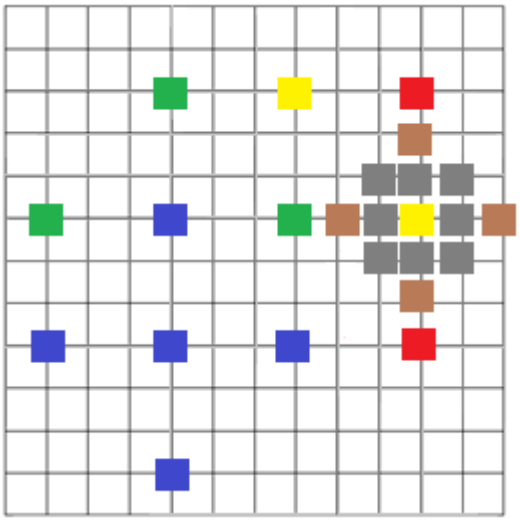

*Figura 9.1: Procesamiento de 2DLOG.*

- ***TSST (Three Step Search Technique).***

Este algoritmo [254] lleva a cabo un proceso que realiza tres iteraciones para así encontrar el macrobloque más parecido dentro de la ventana de búsqueda de la imagen de referencia.

En la primera iteración de TSST, el tamaño del paso de la ventana de búsqueda se designa como la mitad de la zona de búsqueda. A partir de ahí, se seleccionan como en cada iteración nueve puntos candidatos, que son el centro y ocho puntos de comprobación en el límite de la ventana de búsqueda como se muestra en la Figura 9.2 (color azul). La segunda iteración mueve el centro de búsqueda hacia el punto con el mínimo SAD de la etapa anterior, reduciendo el tamaño de paso por la mitad (color verde). En la tercera iteración se detiene el proceso de búsqueda, reduciendo el tamaño del paso a un píxel y obteniendo el vector de movimiento óptimo (color amarillo).

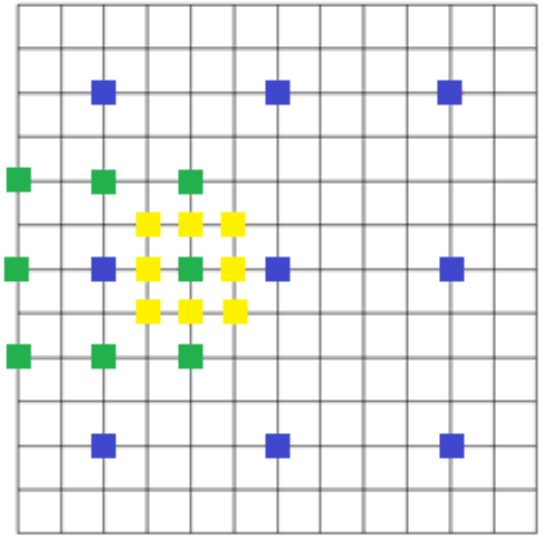

*Figura 9.2: Procesamiento de TSST.*

En la Tabla 9.1, se muestra una comparativa entre FST, TSST, y 2DLOG, mostrando como el FST necesita más puntos de búsqueda que el TSST y a su vez éste más que el 2DLOG.

| Method | Number of Search Points | | Number of Search Steps | |
|---|---|---|---|---|
| | MIN | MAX | MIN | MAX |
| FULL SEARCH (EXHAUSTIVE) | 225 | 225 | 1 | 1 |
| THREE-STEP SEARCH | 25 | 25 | 3 | 3 |
| 2D-LOG SEARCH | 13 | 26 | 2 | 8 |

*Tabla 9.1: Puntos candidatos para FST, TSST, y 2DLOG [214].*

## 9.6. Imagenes y metricas de error utilizadas.

Las imágenes de evaluación seleccionadas en el ámbito multimedia, se presentan en la sección 3.3.1. Usamos tanto formato CIF (352×240) “Caltrain”, “Garden”, “Football” como QCIF (176×144) “Foreman” y “Carphone”. Adicionalmente se presenta en esta sección, las métricas métricas de error usadas en los experimentos de este trabajo, comúnmente utilizadas en el ámbito de codificación y transmisión de video.

### *9.6.1. Métricas de error.*

- SAD (*Sum Of Absolute Differences*): consiste en la suma de cada diferencia entre parejas de puntos separados en tiempo y espacio. Esta métrica ha sido usada en cada etapa de este trabajo para calcular las diferencias entre los distintos macrobloques.

$$SAD(x,y;u,v) = \frac{1}{N^2}\sum_{x=0}^{N-1}\sum_{y=0}^{N-1}\left|I_t(x,y) - I_{t-1}(x+u,y+v)\right| \tag{9.1}$$

Donde $I_t(x,y)$ es el valor del pixel en la coordenada $(x,y)$ en la imagen $t$, $(u,v)$ representa el movimiento del macrobloque candidato y $N$ es el tamaño del macrobloque.

- MSE (*Mean Squared Error*): esta métrica es similar a SAD pero usando diferencias cuadradas lo que le hace menos conservador:

$$MSE(x,y;u,v) = \frac{1}{N^2}\sum_{x=0}^{N-1}\sum_{y=0}^{N-1}\left|I_t(x,y) - I_{t-1}(x+u,y+v)\right|^2 \tag{9.2}$$

$I_t(x, y)$ es el valor del pixel en la coordenada $(x, y)$ en la imagen $t$, $(u, v)$ representa el movimiento del macrobloque candidato y $N$ es el tamaño del macrobloque.

- KPPS (*Kilo Pixels Per Second*): esta métrica se usa para medir el rendimiento del trabajo alcanzado en varias etapas de este trabajo. Esta métrica mide el número de miles de píxeles por segundo que el dispositivo considerado es capaz de procesar.

- PSNR *(Peak Signal to Noise Ratio*): esta métrica se usa para medir la precisión de los algoritmos utilizados, dando una medida cuantitativa de la calidad de la reconstrucción en el ámbito de codificación y transmisión de imágenes:

$$PSNR(x, y, u, v) = 20\log_{10}\left[\frac{Max\_value}{\sqrt{MSE}}\right] \tag{9.3}$$

Donde $Max\_value$ viene dado por la cámara, que en nuestro caso tiene un rango de 8 bits, por lo que su valor es de 256.

## 9.7. FPGAs.

Una FPGA (*Field Programmable Gate Array*) puede ser descrita de forma clásica y general como un chip reprogramable de silicio formado por bloques lógicos previamente construidos y que pueden ser conectados gracias a un enrutamiento configurable. Una FPGA contiene millones de conexiones y celdas lógicas que pueden ser configuradas para lograr un diseño digital específico. En vez de tener un uso restringido a una función hardware predeterminada, la FPGA permite programar las características y funciones del producto, adaptarse a nuevos estándares, y volver a configurar el hardware para aplicaciones específicas, incluso después de que el producto diseñado haya sido instalado.

La FPGAs pueden ser programadas usando una gran variedad de lenguajes de bajo y alto nivel llamados HDL (*Hardware Description Languages*). Debido a la capacidad que poseen las FPGAs para ser configuradas, se puede diseñar hardware personalizado, incluido en un sensor, por ejemplo. Se puede diseñar las características del procesador, desarrollar aceleradores de hardware especializados para tareas de cálculo intensivo, y crear puertos de entrada/salida personalizados para conectar con otra parte física del sistema. Estos sistemas, construidos en la misma FPGA, son hoy en día conocidos como

SoPC (System on a Programmable Chip) [263]. Las FPGAs como se ha apuntado previamente, se configuran en cada nuevo diseño sólo con volver a compilar una configuración diferente del circuito.

La arquitectura FPGA más común se basa en LE (*Logic Elements*), que son la unidad básica, conectados a través de canales reconfigurables que por lo general tienen el mismo número de interconexiones. Los LEs se componen de dos componentes diferentes, flip-flops y LUTs (*Look Up Table*), conectados de diferentes formas dependiendo de la familia de la FPGA. A la hora de proyectar el circuito en una FPGA, el número de elementos lógicos se puede decidir rápidamente, pero el número de interconexiones puede oscilar, incluso entre diseños con la misma cantidad de lógica.

Además de LEs, en una FPGA podemos encontrar multiplicadores empotrados para evitar crearlos usando los LEs de la propia FPGA. Por ejemplo en la Cyclone II, estos bloques pueden realizar multiplicaciones con y sin signo a una velocidad de hasta 250MHz pudiendo ser utilizados en modo de 18×18 bits o de 9×9 bits. Normalmente las FPGAs también tienen bloques empotrados de memoria RAM que proporcionan un alto rendimiento y baja capacidad de almacenamiento. Por ejemplo, la memoria embebida en una Cyclone II consiste en una serie de columnas de memoria M4K que puede funcionar hasta a 250MHz. Cada bloque de M4K tiene una capacidad de almacenamiento de 4068 bits. Estos bloques pueden ser utilizados para implementar diferentes tipos de memoria, incluyendo RAM de un solo puerto, ROM, FIFO (*First In First Out*), tanto para mono como doble puerto y buffers.

### *9.7.1. Procesadores de núcleo blando y duro.*

Podemos clasificar los procesadores en dos principales familias atendiendo al tipo de su núcleo cuando nos centramos en dispositivos FPGA [270]. A continuación presentamos ambos, aunque nuestro trabajo se centra en procesadores *soft-core*.

- Procesadores de núcleo duro (*hard-core*): este tipo de sistemas suelen usar tanto un procesador empotrado dedicado como elementos lógicos de la FPGA. Las FPGAs que usan este tipo de procesadores, presentan una aproximación hibrida entre un ASIC y una FPGA pura. Como ventajas, cabe destacar que este tipo de procesadores presentan

una menor área, una velocidad de reloj mayor, y un rendimiento mayor que los *soft-cores*. Se describen a continuación ejemplos de procesadores hard-core:

o Xilinx PowerPC 405: este procesador [290] de 32 bits presenta una arquitectura PowerPC Harvard RISC *(Reduced Instruction Set Computing*). Podemos encontrarlo en la FPGA Virtex-II Pro en su versión 405D5, y también en las FPGAs Virtex 4 y Virtex 5 usando la versión 405F6.

o ARM (*Advanced RISC Machine*) Cortex-A9 MPCore de doble núcleo: El ARM Cortex A9 MPCore [291] es un procesador de 32 bits desarrollado por ARM Holdings. Como destacable de esta arquitectura, cabe reseñar que usa los 4 primeros bits de la instrucción como código condicional. Además, esta arquitectura presenta la posibilidad de añadir desplazamientos y rotaciones en el procesamiento de datos. Este procesador de doble núcleo es usado por las FPGAs Altera's Arria V [267], Altera's Cyclone V [292], y FPGA Xilix Zinq-7000.

- Procesadores de núcleo blando (*soft-core*): este tipo de procesadores esta construido con los elementos lógicos programables de la FPGA y descrito a través de HDL (*Hardware Description Language*) o lista de conexiones (*netlist*). Al contrario que sus homólogos *hard-cores*, estos procesadores son sintetizados y ajustados dentro de la estructura de la FPGA. Como consecuencia, son muy flexibles y apropiados para la personalización, pudiendo combinar exactamente los microcontroladores con los periféricos, permitiendo incluso configurar la ALU (*Arithmetic Logic Unit*) o el espacio de direcciones de memoria en tiempo de compilación. Son de bajo coste en lo que a integración a nivel de sistema se refiere. Como desventajas, se alcanzan velocidades de reloj más bajas y demandan un mayor consumo de potencia que sus equivalentes con núcleo duro.

Podemos distinguir algunos modelos como Altera Nios II [273], Xilinx Microblaze [284], OpenRISC [294] y Leon4 [293] entre otros. En la Tabla 4.1 del Capítulo 4, presentamos una comparativa donde se ponen de manifiesto las diferencias entre los procesadores de núcleo blando, como unidad de división entera, capacidad de instrucciones a medida, máxima frecuencia de operación, MIPS, LEs y estimación de área para implementar el procesador.

## 9.8. Nios II.

El procesador Nios II [273] de Altera es un procesador *soft-core* y arquitectura RISC, formado por el nucleo del procesador, un grupo de periféricos integrados, y una memoria. Adicionalmente, tiene también una serie de interfaces para memorias no integradas en el chip. Se pueden añadir y eliminar características o periféricos, incluso, en un sistema basado en este procesador puede configurar el mapa de direcciones, el cual está constituido por la dirección de reseteo, la dirección de excepciones, y la dirección del manejador de ruptura.

El núcleo del procesador Nios II está caracterizado poseer un conjunto de instrucciones, ruta de datos, espacio de direcciones y registros de 32-bit. Tiene también 32 tipos de interrupciones, además de instrucciones simples de 32×32 para multiplicar y dividir, adicionalmente hay instrucciones generales en punto flotante. Se tiene acceso a periféricos integrados en el chip e interfaces para memorias y periféricos no integrados en el chip. Tiene modos de depuración asistidos por hardware opcionales como MMU (*Memory Management Unit*) y MPU (*Memory Protection Unit*). Rendimiento de hasta 250 DMIPS (*Dhrystone Millions Instruction Per Second*) y por último instrucciones personalizadas.

### *9.8.1. Estructura hardware del núcleo del Nios II.*

Una vez revisadas las principales características del núcleo del Nios II, vamos a presentar su estructura hardware [273]. Esta arquitectura está compuesta por el núcleo del Nios II y por una serie de periféricos conectados a este núcleo. La arquitectura del núcleo del Nios II se compone de las siguientes unidades funcionales:

- *Program Controller & Address Generation*: este módulo proporciona la generación de las diferentes direcciones de donde las instrucciones del programa son leídas, y el control de programa ejecutado, a través de diferentes componentes internos, aparte de tres señales diferentes para propósitos de reseteo y depuración. Una de estas señales, fuerza al núcleo del procesador para resetearse globalmente de inmediato, otra resetea sólo el núcleo del procesador aunque no otros componentes del sistema Nios II, y la última suspende la ejecución del núcleo del procesador para depuración.

- *Register file*: este módulo consiste en treinta y dos registros internos de propósito general, hasta treinta y dos registros de control y hasta sesenta y tres conjuntos de registros "fantasma" o especiales que son principalmente utilizados para cambios de contexto. Todos los registros son de 32 bits. Además, la arquitectura Nios II permite la adición de futuros registros de punto flotante.

- ALU (*Arithmetic Logic Unit*): la unidad aritmético lógica opera con los datos almacenados en uno o dos registros de propósito general, dependiendo de la entrada, y almacena el resultado calculado en un registro de propósito general. Admite diferentes operaciones, que pueden ser divididas en cuatro familias principalmente: aritméticas, relacionales, lógicas y de desplazamiento/rotación.

Debido a que la arquitectura Nios II admite instrucciones personalizadas, la ALU está conectada directamente a la lógica de la instrucción personalizada, para así acceder al hardware que implementa la misma y usarlo como si fuera una instrucción nativa.

- Controladores de excepción e interrupción: el núcleo del procesador Nios II incluye componentes de hardware para manejo de excepciones con los siguientes componentes:

    - *Exception Controller*: es un controlador simple de excepciones para manejo de todos los tipos de excepciones.

    - EIC (*External Interrupt Controller*): proporciona una interfaz de interrupción hardware de alto rendimiento para reducir la latencia de las interrupciones a través de un controlador externo de interrupciones.

    - IIC (*Internal Interrupt Controller*): gestiona las interrupciones hardware internas.

- Buses de instrucciones y datos: la arquitectura Nios II proporciona diferentes buses para instrucciones y datos, siendo ambos implementados como puertos maestros de la interfaz Avalon-MM. El puerto maestro de instrucciones conecta a los componentes de memoria mientras que el puerto maestro de datos, conecta tanto a la memoria como a los periféricos.

- Cachés de instrucciones y datos: están situadas en los puertos maestros de instrucciones y datos, residiendo en el chip, y mejorando el tiempo de acceso a memorias externas al mismo.

- Interfaces *Tightly-Coupled Memory*: proporcionan interfaces para las memorias "*tightly-coupled*" que se encuentran fuera del núcleo del procesador Nios II. Estas memorias residen en el chip y son similares a las cachés pero sin sobrecarga de cacheo en tiempo real. Además pueden ser usadas tanto, para acceso a instrucciones o datos.

- MMU (*Memory Management Unit*): es opcional, pero cuando se usa, proporciona un mapeo de direcciones virtuales a físicas de 32-bit. Funciona mediante TLBs (*Translation Lookaside Buffer*) en hardware para mejorar la velocidad en el proceso de traducción de direcciones virtuales. Es mutuamente excluyente con el componente MPU (*Memory Protection Unit*).

- MPU (*Memory Protection Unit*): es opcional también, pero cuando se usa, proporciona protección de memoria para hasta 32 regiones de instrucciones de tamaño variable y hasta 32 regiones de datos de tamaño variable. También maneja los permisos de escritura y lectura para las regiones de datos. Es mutuamente excluyente con el componente MMU (*Memory Management Unit*).

- Módulo de debug JTAG: proporciona características para depurar remotamente el núcleo del procesador Nios II desde un host PC. Se usa para descargar a memoria los programas, para arrancar y parar la ejecución, o establecer puntos de ruptura/puntos de vigilancia entre otras tareas. Como principal desventaja, no es soportado por el componente MMU (*Memory Management Unit*).

A continuación mostramos la arquitectura hardware del núcleo del procesador Nios II descrito anteriormente, y un análisis de dicha arquitectura centrándonos en las posibilidades de datos e instrucciones.

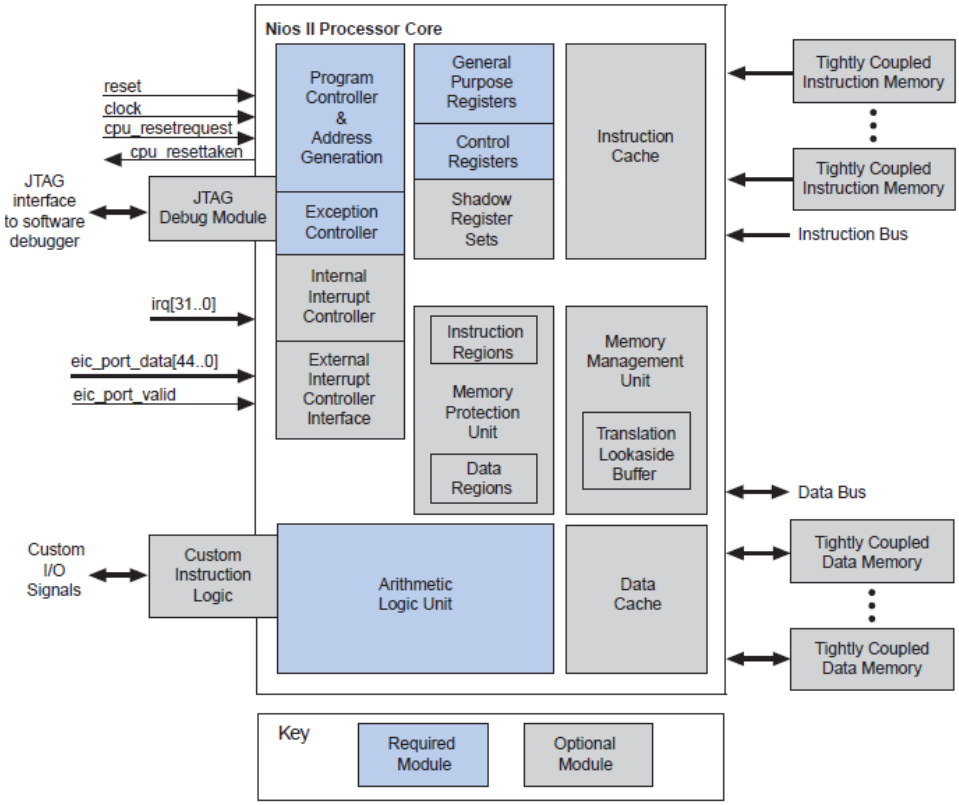


*Figura 9.3: Arquitectura hardware del procesador Nios II [273].*

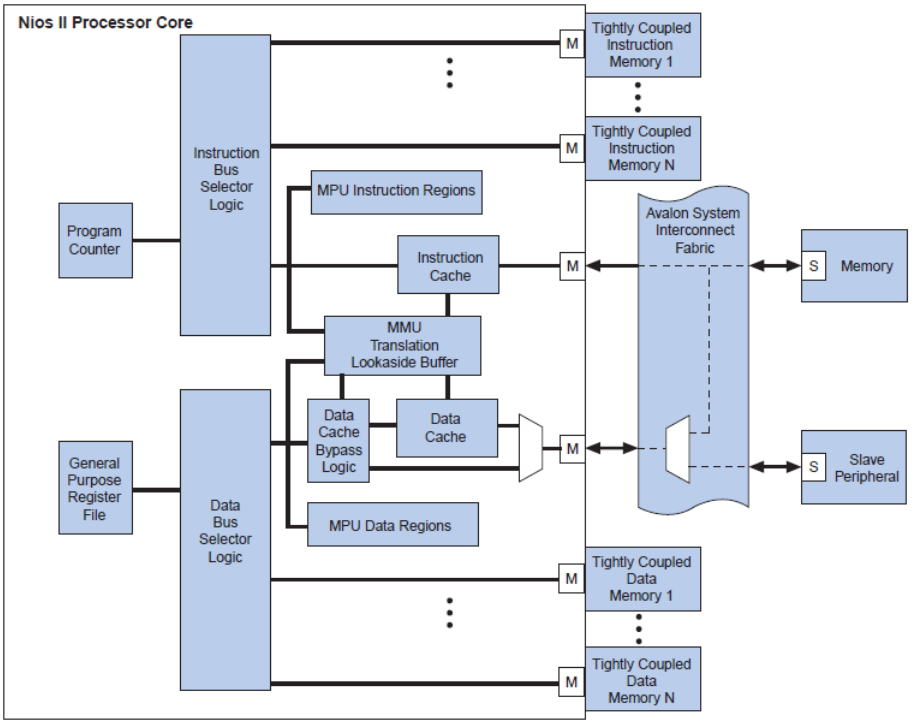


*Figura 9.4: Memoria del Nios II y diagrama de bloques de Entrada/Salida [273].*

### 9.8.2. Tipos del núcleo del procesador Nios II.

El núcleo del procesador Nios II permite tres configuraciones diferentes [273], cada tipo diferente del núcleo del procesador Nios II esta optimizado con una finalidad:

- ***Nios II/e.***

El núcleo del procesador Nios II/e está diseñado para minimizar el uso de recursos hardware pero manteniendo la compatibilidad con el conjunto de instrucciones de la arquitectura del procesador Nios II [273]. De hecho, su tamaño es la mitad del tamaño del núcleo del procesador Nios II/s. Como su principal desventaja, tenemos que esta configuración del núcleo presenta un menor rendimiento comparado con las otras dos configuraciones, como veremos, debido a que está diseñado para aplicaciones de bajo coste. Sus principales características esta descritas a continuación:

- Ejecuta una instrucción en al menos seis ciclos de reloj.

- Permite instrucciones personalizadas.

- El módulo de depuración JTAG está disponible, pero sin soportar puntos de ruptura hardware.

- Emulación software de instrucciones no están implementadas, sin proporcionar soporte hardware.

- Acceso de hasta 2 GB de espacio de direcciones externo.

- Sin cache de instrucciones/datos (cada acceso a memoria/periférico genera una transferencia Avalon-MM).

- No tiene predicción de saltos debido a un pipeline de una sola etapa.

- Circuitería dedicada de desplazamiento (un bit por ciclo) para desplazamiento y rotación.

- Soporte para manejo de excepciones (instrucción ilegal/no implementada, interrupción hardware interna, y excepción software).

- ***Nios II/s.***

El núcleo del procesador Nios II/s está diseñado para minimizar los recursos hardware utilizados pero mejorando el rendimiento de ejecución [273]. Su rendimiento de ejecución es alrededor del 40% inferior al del núcleo del procesador Nios II/f aunque usando un 20% menos de sus recursos hardware. Este núcleo está diseñado a modo de compromiso entre coste y rendimiento. Sus principales características se describen a continuación:

- Ejecuta una sola instrucción en un ciclo de reloj (instrucciones normales de la ALU) o más.

- Disponibilidad de caché de instrucciones con mapeo directo y tamaño definido por el usuario entre 512 bytes y 64 KB, capaz de leer una línea entera de la caché cada vez.

- Acceso de hasta 2 GB de espacio de direcciones externo.

- Permite instrucciones personalizadas.

- Sin cache de datos, este hecho será importante como veremos en las siguientes secciones experimentales de este trabajo.

- Pipeline de cinco etapas (*Fetch*, *Decode*, *Execute*, *Memory*, y *Writeback*).

- Predicción de saltos estática usando desplazamientos de dirección para los saltos (negativo para salto hacia atrás y positivo para salto hacia adelante).

- Proporciona opciones de multiplicación por hardware (multiplicadores DSP, multiplicadores embebidos, y multiplicadores construidos con LEs), opciones de división y opciones de rotación (tres o cuatro ciclos cuando el multiplicador de hardware existe, o un bit por ciclo con circuitería dedicada).

- Modulo de depuración JTAG disponible incluyendo puntos de ruptura hardware y trazas en tiempo real.

- Soporte para manejo de excepciones (instrucción ilegal/no implementada, interrupción hardware interna, y excepción software).

- Soporte para hasta 4 memorias de tipo "*tightly-coupled*" para instrucciones, evitando la memoria caché de instrucciones, usando cada una un puerto maestro de la interfaz de memoria y conectada directamente a un puerto esclavo de la memoria.

- El puerto maestro de datos está siempre presente, sin embargo el puerto maestro de instrucciones solo está incluido cuando lo está también la caché de instrucciones.

- ***Nios II/f.***

El núcleo del procesador Nios II/f está diseñado para un alto rendimiento de ejecución conseguido a través de un gran tamaño del núcleo. De hecho, si comparamos su tamaño con el del núcleo del procesador Nios II/s, podemos ver que su tamaño es cerca de un 25% superior. Dado que es el núcleo del procesador Nios II con el mayor rendimiento, está diseñado para aplicaciones sensibles al rendimiento. Sus principales características están descritas a continuación:

- Permite instrucciones personalizadas.

- Acceso de hasta 2 GB de espacio de direcciones externo cuando no está presente el módulo MMU, y de hasta 4 GB cuando si lo está.

- Hasta 63 conjuntos de registros fantasma opcionales.

- Predicción de saltos dinámica usando una tabla histórica de saltos de dos bits.

- Proporciona opciones de multiplicación por hardware (multiplicadores DSP, multiplicadores embebidos, y multiplicadores construidos con LEs), opciones de división, y opciones de rotación (uno o dos ciclos cuando el multiplicador de hardware existe, o un bit por ciclo con circuitería dedicada).

- Modulo de depuración JTAG disponible incluyendo punto de ruptura hardware y trazas en tiempo real excepto para el módulo MMU.

- Puertos maestros opcionales para instrucciones (caché de instrucciones habilitada) y datos (caché de datos habilitada).

- Ejecuta una sola instrucción en un ciclo (mayoría de instrucciones de la ALU) o más.

- Disponibilidad de caché de instrucciones con mapeo directo y tamaño definido por el usuario entre 512 bytes y 64 KB, capaz de leer una línea entera de la caché (32 bytes) en un solo ciclo de reloj. Está virtualmente indexada cuando el módulo MMU está presente.

- Disponibilidad de caché de datos con mapeo directo y tamaño definido por el usuario entre 512 bytes y 64 KB, capaz de leer una línea entera de la caché (configurable a 4, 16, ó 32 bytes) en un solo ciclo de reloj. Está virtualmente indexada cuando el módulo MMU está presente.

- Soporte para memorias de tipo "*tightly-coupled*" tanto para instrucciones como para datos evitando las cachés cuando están habilitadas, usando cada una un puerto maestro de la interfaz de memoria, y conectada directamente a un puerto esclavo de la memoria.

- Pipeline de seis etapas (*Fetch*, *Decode*, *Execute*, *Memory*, *Align*, y *Writeback*).

- Módulo MMU (*Memory Management Unit*) opcional que proporciona un TLB (*Translation Lookaside Buffer*) que consiste en un TLB principal, un micro TLB para instrucciones (µITLB), y otro micro TLB para datos (µDTLB), siendo ambos almacenados a través de registros basados en LEs. Los µTLB se usan como cache del TLB principal y no están visibles a través de software.

- Módulo MPU (*Memory Protection Unit*) opcional.

- Soporte para manejo de excepciones (instrucción ilegal/no implementada, interrupción hardware interna, excepción software, desalineación, error de división, excepción del TLB, violación de región del MPU, y supervisor de instrucción/dirección de instrucción/dirección de datos).

- Interfaz opcional EIC (*External Interrupt Controller*) para impulsar el manejo de interrupciones a través de diferentes señales (RHA (*Requested Handler Address*), RRS (*Requested Register Set*), RIL (*Requested Interrupt Level*), y RNMI (*Requested Nonmaskeable Interrupt*)).

## 9.9. DE2-C35.

Después de haber descrito las diferentes arquitecturas del Nios II, se procede a mostrar la placa sobre la cual va alojada la FPGA de bajo consumo [274 - 275] que ha sido usada para los experimentos de este trabajo. La placa en cuestión, DE2-C35, es una tarjeta diseñada para el aprendizaje bajo requisitos de bajo coste [275] que contiene una FPGA Cyclone II EP2C35 (672 pines) conectada a diferentes periféricos. En la Figura 9.5 mostramos una imagen de la placa DE2-C35. Presentando, adicionalmente en la Figura 9.6 un diagrama de bloques de dicha placa con las diferentes características, funcionalidades, y periféricos conectados.

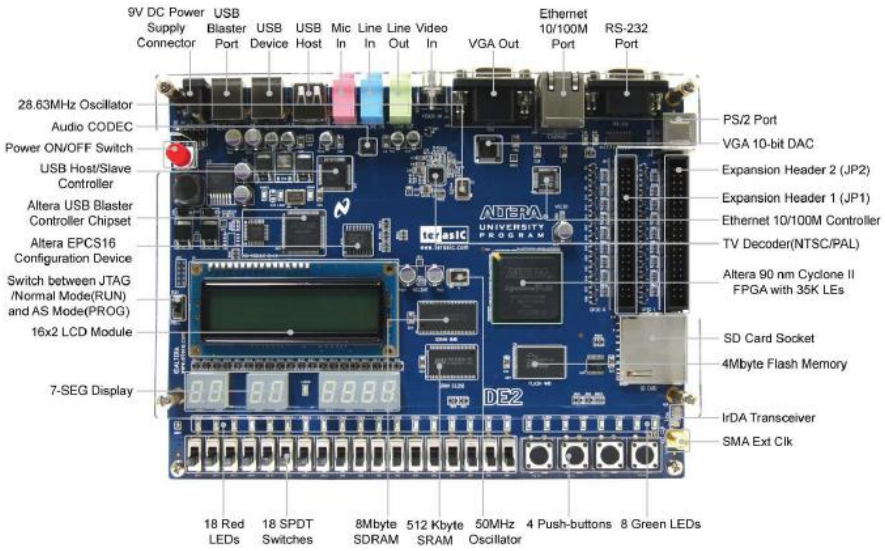


*Figura 9.5: Placa DE2-C35 [275].*

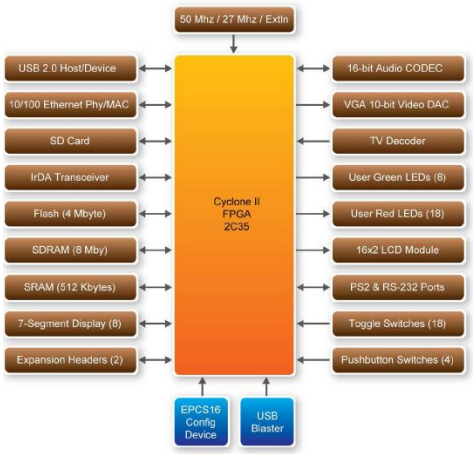


*Figura 9.6:Diagrama de bloques de la placa DE2-C35 [275].*

## 9.10. Introducción a C2H.

En esta sección, vamos a acelerar los algoritmos presentados usando el paradigma C2H (C to Hardware) de Altera. El compilador C2H [295] es una herramienta disponible desde Altera [304], siendo su tarea la de construir aceleradores hardware tomando como entrada código ANSI C [302], mejorando el rendimiento de los programas ejecutados en el procesador Nios II. Esto es debido a una sustitución de las funciones de software elegidas por aceleradores hardware pertinentes dentro del diseño descargado a la FPGA [307].

Cada función elegida para ser acelerada será traducida a un acelerador hardware siempre y cuando dicha función cumpla con los requisitos del compilador C2H, como estar definida con un subconjunto de código ANSI C soportado, siendo esta circunstancia su principal inconveniente [295]. Estos aceleradores hardware son añadidos gracias a la herramienta SOPC Builder, que inserta en el diseño del procesador Nios II el acelerador a través del módulo Avalon [298 – 299], al igual que otros periféricos del sistema. Estos aceleradores llevan inmersas diferentes características como el acceso directo a memoria.

A continuación, mostramos el proceso realizado por el C2H para la generación de un acelerador hardware con objeto de alcanzar una mejora óptima de rendimiento.

1) El preprocesador GCC [305] evalúa las directivas.
2) Se procede al "*parsing*" del código pre-procesado para proporcionar una serie de objetos, como un compilador estándar.
3) Creación de un grafo de dependencias [295] con los objetos resultantes del paso anterior para transformarlo en una máquina de estados que gestiona la secuencia de operaciones definida por el código fuente y que finalmente será acelerada.
4) Optimizaciones: primero, la herramienta C2H convierte directamente cada elemento del código fuente en una estructura hardware equivalente usando reglas sencillas de traducción. Después, el compilador C2H realiza diferentes tareas de optimización para reducir el uso de elementos lógicos a través del uso de recursos compartidos, alcanzando un mejor resultado que en un mapeo uno a uno.
5) Mejor secuencia: en este paso, el compilador C2H busca la mejor secuencia para realizar cualquier operación en la función acelerada.

6) Generación del acelerador hardware: aquí, el C2H genera un fichero objeto sintetizable HDL [308] que define el acelerador hardware generado.

7) Generar un envoltorio (*wrapping*) software [295] para controlar e interactuar con el acelerador hardware generado, leyendo y escribiendo a través de la interfaz de registros. De hecho, desde la perspectiva de la llamada, la funcionalidad es totalmente idéntica a la de llamar a la función original.

En la Figura 9.7 se muestra como se incluye en el sistema basado en el procesador Nios II un acelerador hardware generado por la herramienta C2H. El módulo de Altera Avalon switch fabric [298 – 299] actúa como un bus estándar que interconecta los diferentes componentes en un sistema basado en el procesador Nios II, permitiendo la transferencia de datos entre ellos. La interfaz Avalon usada para interconectar el acelerador hardware con los diferentes componentes del sistema es del tipo Avalon-MM (*Memory Mapped*) [298]. Esta interfaz sigue el paradigma maestro-esclavo, donde los maestros pueden iniciar la transacción, y los esclavos responden a las peticiones de los maestros generando los datos de vuelta. El bloque MUX (Multiplexor) permite seleccionar datos desde el esclavo deseado mientras los árbitros del bus deciden que maestro accede al control en cada caso.

Cabe destacar adicionalmente, como consecuencia los aceleradores hardware consumen recursos de la FPGA, de forma que tanto los elementos lógicos como la memoria On-chip consumida no están disponibles para los demás componentes del diseño.

## 9.11. Arquitecturas propuestas.

A continuación, mostramos los algoritmos que serán acelerados con la herramienta C2H. De hecho para presentar bien los algoritmos, vamos a mostrar su flujo de datos y el de las funciones internas a estos. En la Figuras 9.15, 9.16 y 9.17 se muestran respectivamente el flujo de datos para el algoritmo FST, TSST y 2DLOG.

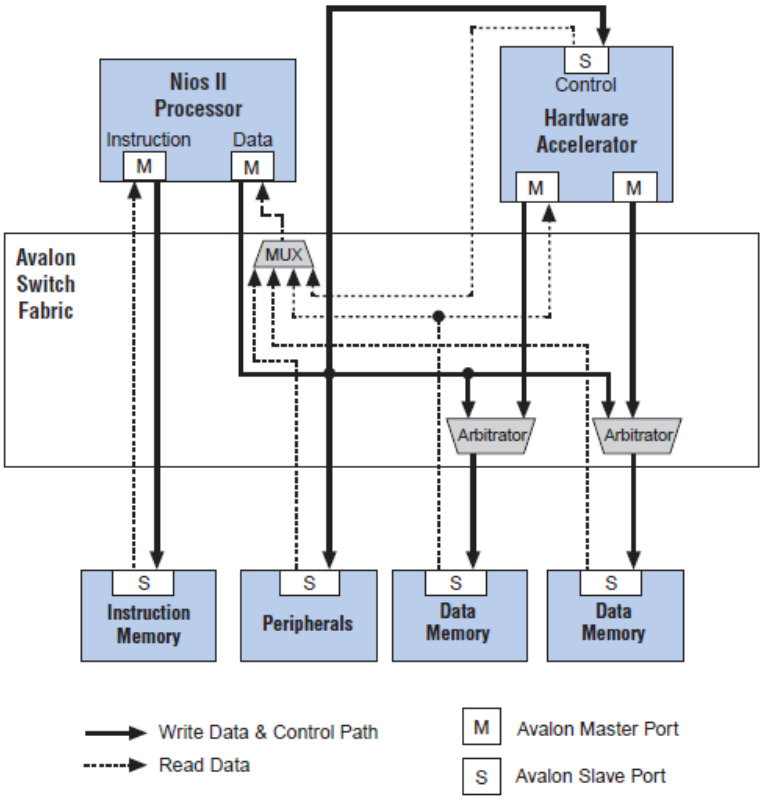


*Figura 9.7: Sistema ejemplo con un acelerador hardware [295].*

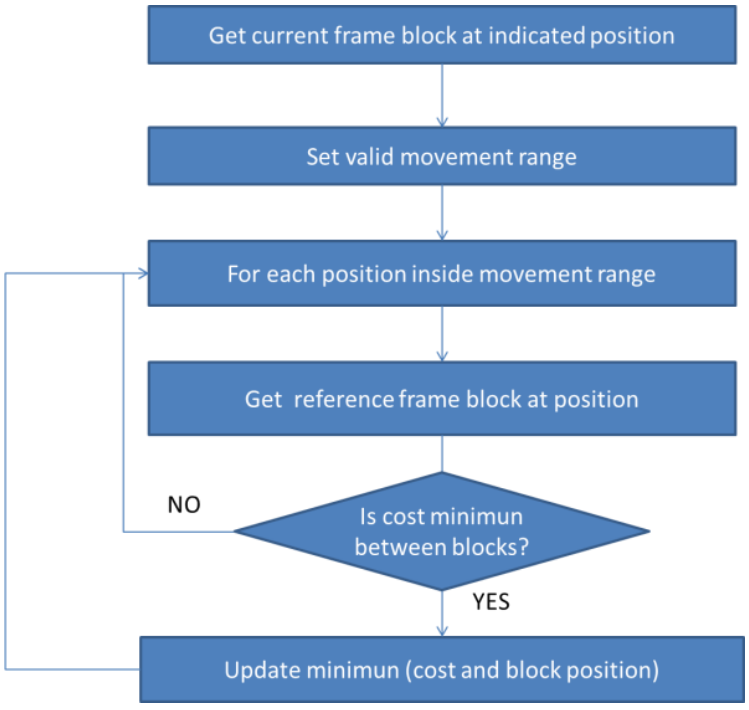


*Figura 9.8: Flujo de datos de FST [297].*

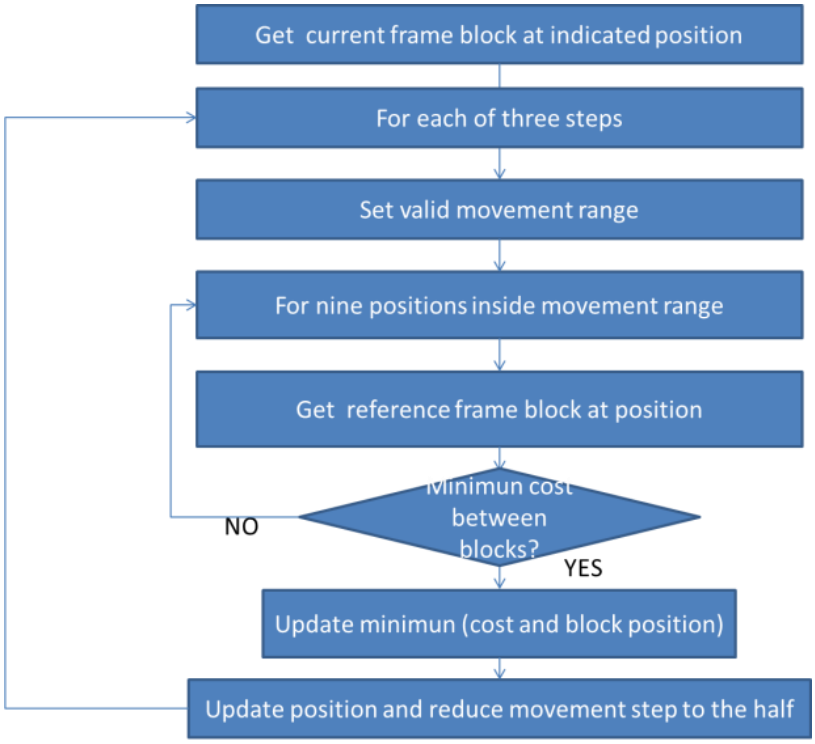


*Figura 9.9: Flujo de datos de TSST [297].*

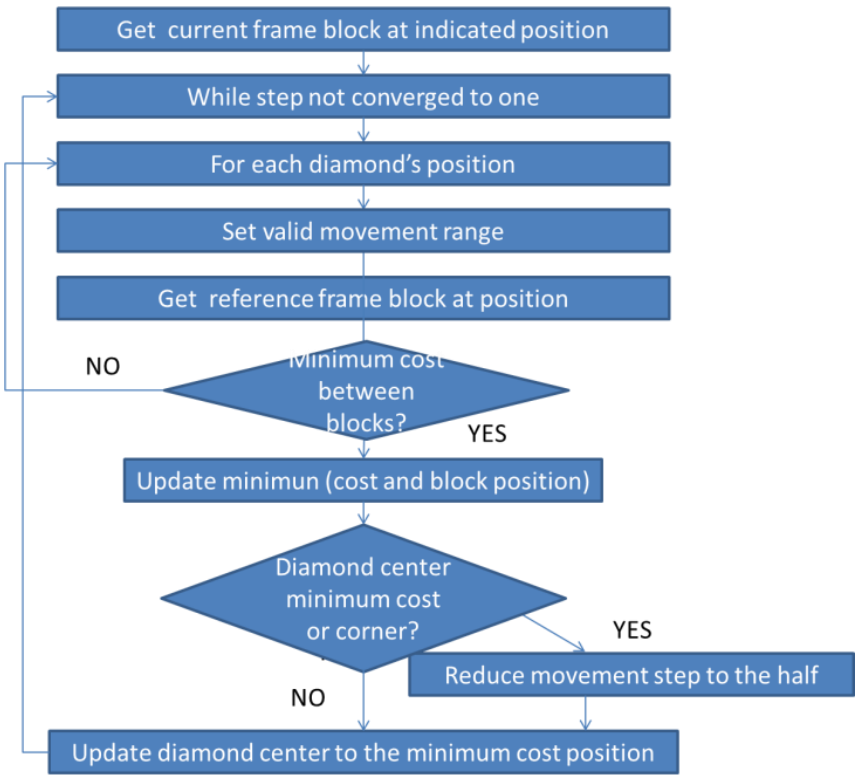


*Figura 9.10: Flujo de datos de 2DLOG [297].*

Debido a que nuestro trabajo está orientado respecto a bajo coste y consumo, buscamos un balance entre los recursos hardware usados y el rendimiento obtenido. Por esta razón, hemos organizado los algoritmos presentados en diferentes funciones, para clasificar la aceleración de los algoritmos en diferentes calidades. Dichas funciones y sus flujos de datos se encuentran descritos a continuación:

- CopyBlock: esta función copia un número fijo de bytes de la dirección origen a la dirección destino., siendo su flujo de datos el siguiente:

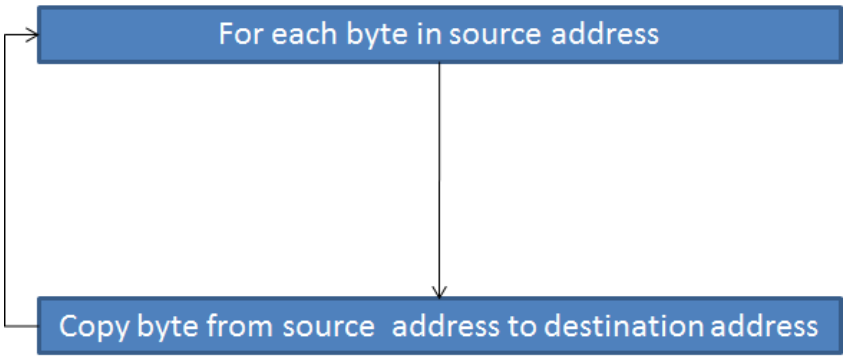


*Figura 9.11: Flujo de datos de CopyBlock [297].*

- GetBlock: esta función copia un macrobloque de la dirección origen a la dirección destino, siendo su flujo de datos el siguiente:

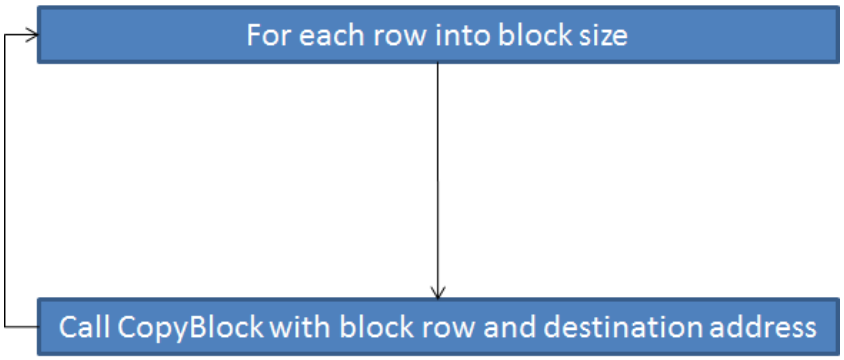


*Figura 9.12: Flujo de datos de GetBlock [297].*

- GetCost: esta función calcula el coste entre dos macrobloques para una métrica dada, siendo su flujo de datos el siguiente:

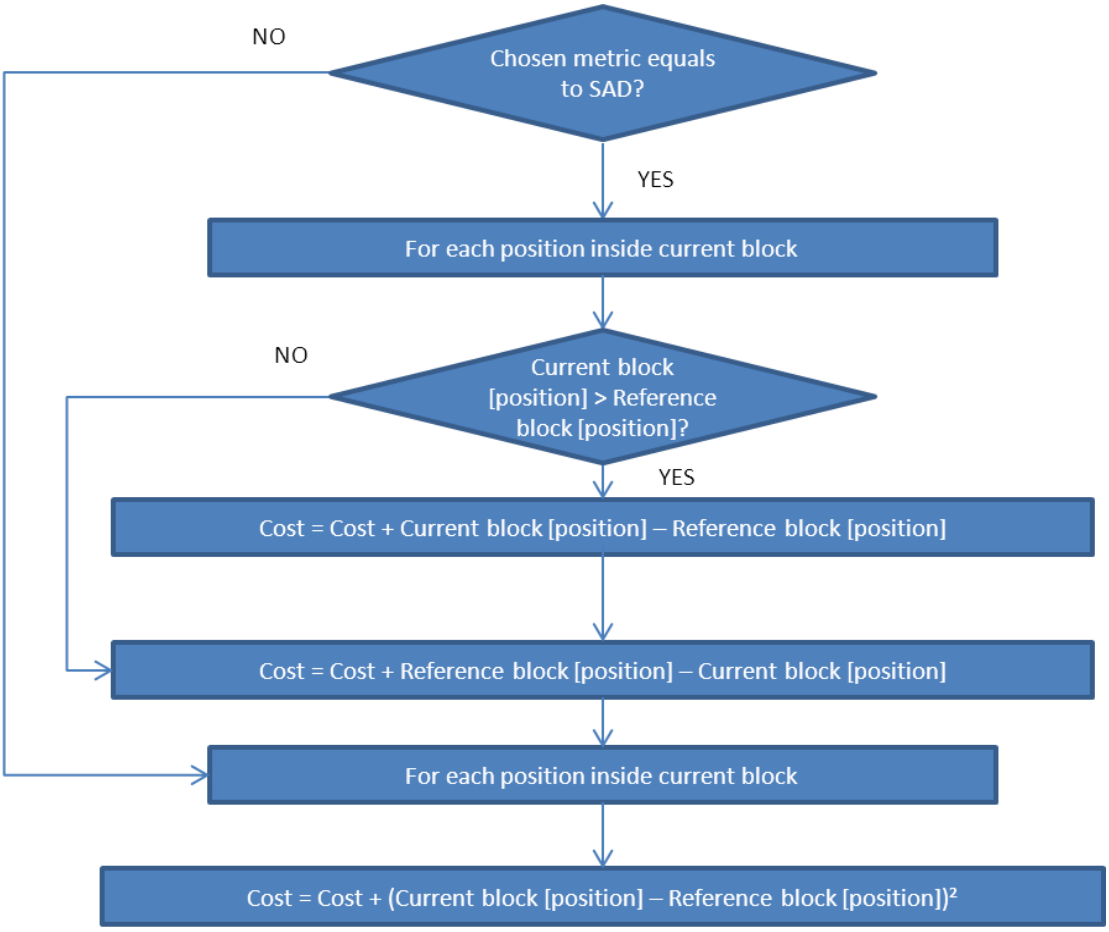


*Figura 9.13: Flujo de datos de GetCost [297].*

### 9.11.1. Evaluación de tiempos y llamadas (Profiling).

Para saber cuál es el punto donde se produce la mayor pérdida de tiempo de ejecución en nuestros tres algoritmos (FST, TSST, y 2DLOG), hemos realizado una evaluación completa de cada uno de los algoritmos presentados. Este perfilado ha sido realizado para nuestros principales estímulos ("Foreman" y "Carphone" presentados en la Sección 3.3.1) usando tanto cada tamaño de macrobloque disponible (16×16, 32×32, y 64×64), como cada tamaño de ventana disponible (8×8, 16×16, y 32×32).

Se presentan los resultados indicados usando la secuencia "Foreman" y"Carphone", para las Tablas 9.2 – 9.4 y 9.5 – 9.7. Se muestran el número de llamadas a cada función y el porcentaje de tiempo de ejecución empleado en dichas llamadas, denominando las funciones de la siguiente manera: CB (CopyBlock), GB (GetBlock), GC (GetCost), y FN (FST, 2DLOG, o TSST dependiendo del algoritmo ejecutado).

| Tamaño de ventana | Macrobloque de 16 | | | | Macrobloque de 32 | | | | Macrobloque de 64 | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | **CB** | **GB** | **GC** | **FN** | **CB** | **GB** | **GC** | **FN** | **CB** | **GB** | **GC** | **FN** |
| 8 | 376416 (12.5) | 23526 (~0) | 23427 (87.5) | 99 (~0) | 148800 (16.67) | 4650 (~0) | 4620 (83.33) | 30 (~0) | 43480 (~0) | 685 (~0) | 676 (~100) | 9 (~0) |
| 16 | 1405024 (7.41) | 87814 (7.41) | 87715 (85.19) | 99 (~0) | 562560 (4.35) | 17580 (~0) | 17550 (95.65) | 30 (~0) | 163776 (8.33) | 2559 (~0) | 2550 (91.67) | 9 (~0) |
| 32 | 4844640 (8.57) | 302790 (0.48) | 302691 (90.95) | 99 (~0) | 2155392 (8.57) | 67356 (1.43) | 67326 (90.00) | 30 (~0) | 604096 (14.63) | 9439 (~0) | 9430 (85.37) | 9 (~0) |

*Tabla 9.2: Perfil de FST [320].*

| Tamaño de ventana | Macrobloque de 16 | | | | Macrobloque de 32 | | | | Macrobloque de 64 | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | **CB** | **GB** | **GC** | **FN** | **CB** | **GB** | **GC** | **FN** | **CB** | **GB** | **GC** | **FN** |
| 8 | 26384 (~0) | 1649 (~0) | 1550 (~100) | 99 (~0) | 10784 (~0) | 337 (~0) | 307 (~100) | 30 (~0) | 3968 (~0) | 62 (~0) | 53 (~100) | 9 (~0) |
| 16 | 35616 (~0) | 2226 (~0) | 2127 (~100) | 99 (~0) | 14016 (~0) | 438 (~0) | 408 (~100) | 30 (~0) | 4992 (~0) | 78 (~0) | 69 (~100) | 9 (~0) |
| 32 | 43280 (~0) | 2705 (~0) | 2606 (~100) | 99 (~0) | 25952 (~0) | 811 (~0) | 781 (~100) | 30 (~0) | 8640 (~0) | 135 (~0) | 126 (~100) | 9 (~0) |

*Tabla 9.3: Perfil de 2DLOG [320].*

| Tamaño de ventana | Macrobloque de 16 | | | | Macrobloque de 32 | | | | Macrobloque de 64 | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | **CB** | **GB** | **GC** | **FN** | **CB** | **GB** | **GC** | **FN** | **CB** | **GB** | **GC** | **FN** |
| 8 | 38976 (~0) | 2436 (~0) | 2337 (~100) | 99 (~0) | 19040 (~0) | 595 (~0) | 565 (~100) | 30 (~0) | 8640 (~0) | 135 (~0) | 126 (~100) | 9 (~0) |
| 16 | 38928 (~0) | 2433 (~0) | 2334 (~100) | 99 (~0) | 19040 (~0) | 595 (~0) | 565 (~100) | 30 (~0) | 8384 (~0) | 131 (~0) | 122 (~100) | 9 (~0) |
| 32 | 38832 (~0) | 2427 (~0) | 2328 (~100) | 99 (~0) | 20480 (~0) | 640 (~0) | 610 (~100) | 30 (~0) | 8640 (~0) | 135 (~0) | 126 (~100) | 9 (~0) |

*Tabla 9.4: Perfil de TSST [320].*

| Tamaño de ventana | Macrobloque de 16 | | | | Macrobloque de 32 | | | | Macrobloque de 64 | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | CB | GB | GC | FN | CB | GB | GC | FN | CB | GB | GC | FN |
| 8 | 376416 (14.29) | 23526 (~0) | 23427 (85.71) | 99 (~0) | 148800 (6.67) | 4650 (~0) | 4620 (93.33) | 30 (~0) | 43480 (~0) | 685 (~0) | 676 (~100) | 9 (~0) |
| 16 | 1405024 (13.04) | 87814 (~0) | 87715 (86.96) | 99 (~0) | 562560 (11.76) | 17580 (~0) | 17550 (88.24) | 30 (~0) | 163776 (~0) | 2559 (~0) | 2550 (~100) | 9 (~0) |
| 32 | 4844640 (3.57) | 302790 (2.86) | 302691 (93.57) | 99 (~0) | 2155392 (9.32) | 67356 (~0) | 67326 (90.68) | 30 (~0) | 604096 (14.29) | 9439 (2.86) | 9430 (82.86) | 9 (~0) |

*Tabla 9.5: Perfil de FST [320].*

| Tamaño de ventana | Macrobloque de 16 | | | | Macrobloque de 32 | | | | Macrobloque de 64 | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | CB | GB | GC | FN | CB | GB | GC | FN | CB | GB | GC | FN |
| 8 | 26464 (~0) | 1654 (~0) | 1555 (~100) | 99 (~0) | 10624 (~0) | 332 (~0) | 302 (~100) | 30 (~0) | 3648 (~0) | 57 (~0) | 48 (~100) | 9 (~0) |
| 16 | 34256 (~0) | 2141 (~0) | 2042 (~100) | 99 (~0) | 13696 (~0) | 428 (~0) | 398 (~100) | 30 (~0) | 4672 (~0) | 73 (~0) | 64 (~100) | 9 (~0) |
| 32 | 41088 (~0) | 2568 (~0) | 2469 (~100) | 99 (~0) | 25312 (~0) | 791 (~0) | 761 (~100) | 30 (~0) | 8256 (~0) | 129 (~0) | 120 (~100) | 9 (~0) |

*Tabla 9.6: Perfil de 2DLOG [320].*

| Tamaño de ventana | Macrobloque de 16 | | | | Macrobloque de 32 | | | | Macrobloque de 64 | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | CB | GB | GC | FN | CB | GB | GC | FN | CB | GB | GC | FN |
| 8 | 39072 (~0) | 2442 (~0) | 2343 (~100) | 99 (~0) | 18944 (~0) | 592 (~0) | 562 (~100) | 30 (~0) | 8576 (~0) | 134 (~0) | 125 (~100) | 9 (~0) |
| 16 | 38928 (~0) | 2433 (~0) | 2334 (~100) | 99 (~0) | 18944 (~0) | 592 (~0) | 562 (~100) | 30 (~0) | 8576 (~0) | 134 (~0) | 125 (~100) | 9 (~0) |
| 32 | 38320 (~0) | 2395 (~0) | 2296 (~100) | 99 (~0) | 20384 (~0) | 637 (~0) | 607 (~100) | 30 (~0) | 8896 (~0) | 139 (~0) | 130 (~100) | 9 (~0) |

*Tabla 9.7: Perfil de TSST [320].*

Los resultados del perfil respecto del código original sugieren que la función GetCost es la más apropiada para ser acelerada, como se puede apreciar en las tablas anteriores.

### *9.11.2. Clasificación de las calidades de aceleración.*

Para dar elección al desarrollador entre consumo y rendimiento, hemos dividido la calidad de la aceleración en las cuatro categorías siguientes:

- No [aceleración]: todo el código fuente es ejecutado como software y no se crea ni coloca ningún acelerador hardware.

- Low: solo la función CopyBlock es ejecutada a través de un acelerador hardware creado con la herramienta C2H.

- Medium: las funciones CopyBlock y GetCost son ejecutadas a través de aceleradores hardware creados con la herramienta C2H.

- High: todas las funciones presentadas son ejecutadas por sus respectivos aceleradores hardware generados por la herramienta C2H.

## 9.12. Resultados obtenidos para las arquitecturas presentadas.

En esta sección, presentamos los resultados obtenidos con las calidades de aceleración descritas anteriormente usando como entrada las secuencias conocidas [227] ya descritas en la Sección 3.3.1.

### *9.12.1. Resultados medidos en KPPS (Kilo Pixels Per Second).*

Los KPPS se usan como medida de rendimiento para un sistema completo, usando SAD como la métrica de error en la ejecución de los algoritmos presentados. En la Figura 9.14 mostramos los resultados obtenidos para cada algoritmo (FST, 2DLOG y TSST), cada tamaño de ventana (8, 16 y 32), cada calidad de aceleración (No, Low, Medium, y High), y cada tipo de procesador de Nios II disponible (Economic (Nios II/e), Standard (Nios II/s) and Fast (Nios II/f)).

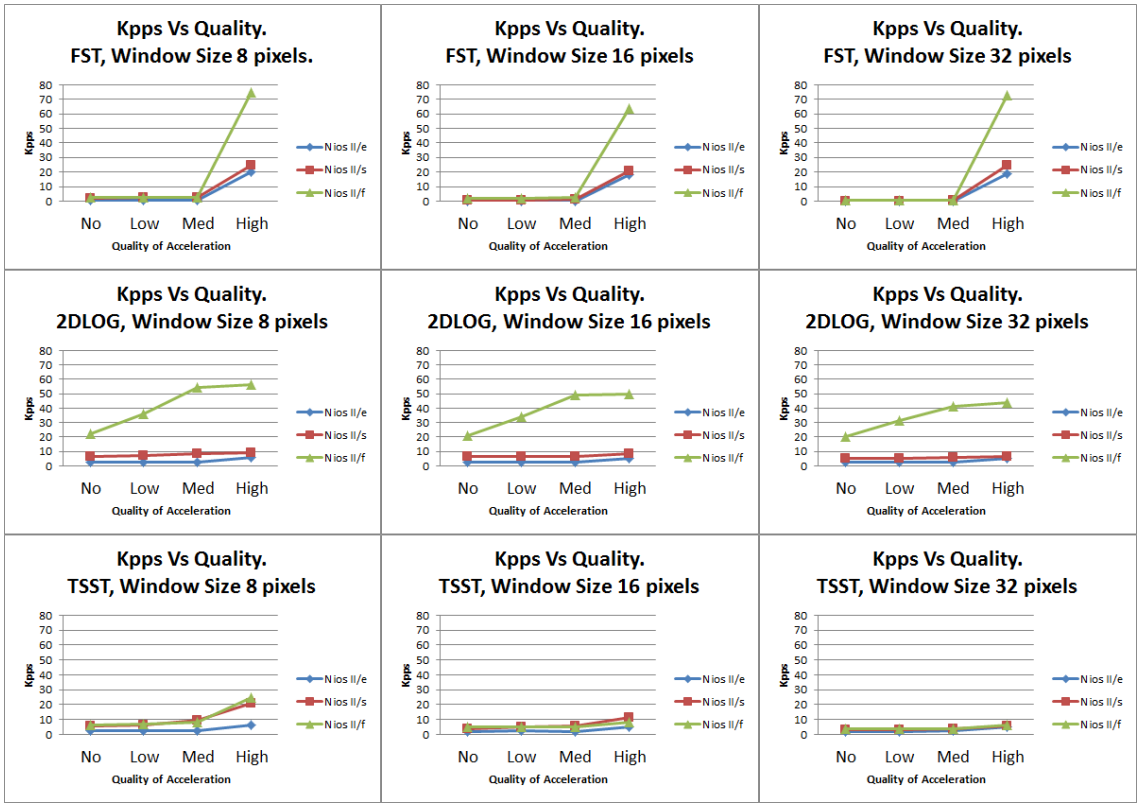


*Figura 9.14: Rendimiento medido en KPPS [297].*

El rendimiento del sistema con el algoritmo FST se comporta de manera similar con aceleraciones No, Low, y Medium. Si nos centramos en el algoritmo TSST, este comportamiento se convierte en lineal (con aceleraciones No, Low, y Medium). Si nos fijamos en la técnica 2DLOG, la respuesta lineal se aprecia de forma nítida una vez más para las aceleraciones No, Low, y Medium, especialmente en la arquitectura (Nios II/f). Aunque el rendimiento de cada arquitectura depende del tamaño de la ventana (8, 16, y 32 píxeles), se muestra la tendencia lineal de las respuestas para todos los tamaños.

Por ejemplo, usando aceleración Medium con 2DLOG y Nios II/f se obtiene un rendimiento entre 16 y 21 fps con una resolución de 50×50 píxeles y diferentes tamaños de ventana, y un rendimiento mayor al usar aceleración High en cada tamaño de ventana.

Centrándonos en el FST bajo arquitectura Nios II/f, obtenemos de 61 a 72 KPPS dependiendo del tamaño de la ventana utilizada (desde 8 hasta 32 píxeles). Esto corresponde a un rendimiento del sistema de entre 24.5 y 29 fps a una resolución de 50×50 píxeles, suficiente para un pequeño sensor. Para las configuraciones con la

arquitectura de Nios II/e o Nios II/s, se obtiene una gama de rendimiento de entre 20 y 27 KPPS (desde 8 hasta 32 píxeles), que corresponde a un rango de entre 8 y 11 fps con una resolución de 50×50 píxeles.

En cuanto al TSST bajo la arquitectura Nios II/f, obtenemos un rango de 6.15 a 24.6 KPPS, en función del tamaño de la ventana utilizada (desde 8 hasta 32 píxeles). Esto significa un rendimiento para el sistema de entre 2 y 10 fps con una resolución de 50×50 píxeles. Para las configuraciones que utilizan Nios II/e o Nios II/s, se obtiene un rango entre 5 y 20 KPPS (desde 8 hasta 32 píxeles) lo que significa entre 2 y 8 fps para una resolución de 50×50 píxeles.

Por último, el algoritmo 2DLOG procesa un rango de 43.8 a 56.4 KPPS bajo la arquitectura Nios II/f, de nuevo dependiendo del tamaño de la ventana utilizada (desde 8 hasta 32 píxeles), lo que significa una gama de 17.5 a 22.5 fps. Para las configuraciones en el bajo Nios II/e y Nios II/s, el rendimiento alcanzado es entre 8 y 10 KPPS, independientemente del tamaño de ventana (desde 8 hasta 32 píxeles), lo que significa un rango de entre 2 y 8 fps en una resolución de 50×50 píxeles.

Conviene destacar que el tamaño de la ventana no siempre es inversamente proporcional al rendimiento del sistema. A modo de ejemplo, el TSST restringe la complejidad de cálculo mediante la limitación de la búsqueda exhaustiva en tres pasos, por lo que la aceleración de todas las funciones pone de manifiesto una solución de compromiso entre el nivel paralelo de píxeles, y el incremento del tamaño del macrobloque.

### *9.12.2. Resultados medidos en PSNR (Peak Signal to Noise Ratio).*

PSNR se utiliza como una medida del rendimiento usando el MSE (Mean Squared Error) en lugar del SAD, debido al hecho de que el MSE es menos conservador y enfatiza las diferencias más grandes. En la Figura 9.15, se presentan los resultados obtenidos para las secuencias “Caltrain”, “Garden”, y “Football”, con una resolución de 352×240 (4:2:0 y formato CIF).

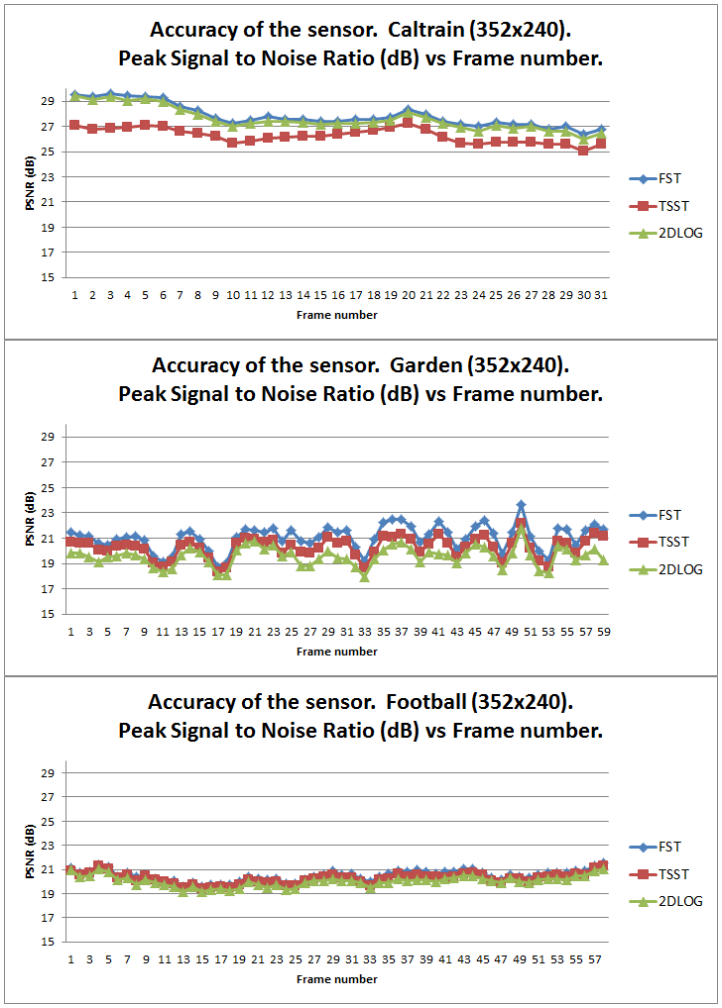


*Figura 9.15: Precisión medida en PSNR [297].*

La precisión del FST es la más alta de los tres algoritmos presentados debido a su búsqueda exhaustiva, mientras que las otras dos técnicas, TSST y 2DLOG, se alternan en términos de precisión. Los resultados obtenidos muestran una diferencia menor a 2 dB (decibelios) entre las tres implementaciones presentadas para cada secuencia.

### *9.12.3. Resultados medidos en recursos hardware utilizados.*

En la Tabla 9.8, se muestran los recursos de hardware utilizados (*Logic Cells*, EMS (*Embedded Multiplier*), y *Total Memory bits*) para cada técnica (FST, 2DLOG, y TSST), para un tamaño de ventana fijado a 32 píxeles y una aceleración High, bajo los tres procesadores: Nios II/e, Nios II/s, y Nios II/f. El tamaño de la ventana se fija a 32 por ser el más costoso y presentar los tiempos más lentos. La aceleración se ha fijado a High debido a que se necesita más recursos hardware que las otras aceleraciones.

| Nios II/ Quality | Method | Logic Cells | Method | Logic Cells | Method | Logic Cells | (FST,TSST,2DLOG) EMs(9× 9) | (FST,TSST,2DLOG) Total memory bits |
|---|---|---|---|---|---|---|---|---|
| e / High | FST | 11637 (35%) | TSST | 13173 (40%) | 2DLOG | 13056 (39%) | 23 (33%) | 44032 (9%) |
| s / High | | 12382 (37%) | | 14023 (42%) | | 13955 (42%) | 27 (39%) | 79488 (16%) |
| f / High | | 13090 (39%) | | 14755 (44%) | | 14678 (44%) | 27 (39%) | 114944 (24%) |

*Tabla 9.8: Recursos hardware utilizados con tamaño de ventana de 32 [297].*

En la Tabla 9.9, se muestran recursos hardware utilizados (*Logic Cells*, EMS (*Embedded Multiplier*), y *Total Memory bits*) para FST, 2DLOG o TSST, con un tamaño de ventana de 32 píxeles, bajo los tres procesadores: Nios II /e/s/f. Combinamos estas configuraciones con las otras tres aceleraciones presentadas, No, Low, y Medium.

| Quality | Nios II/e | | | Nios II/s | | | Nios II/f | | |
|---|---|---|---|---|---|---|---|---|---|
| | Logic Cells | EMs (9 × 9) | Total memory bits | Logic Cells | EMs (9 × 9) | Total memory bits | Logic Cells | EMs (9 × 9) | Total memory bits |
| No | 2107 (6%) | 0 (0%) | 44032 (9%) | 3085 (9%) | 4 (6%) | 79488 (16%) | 3763 (11%) | 4 (6%) | 114944 (24%) |
| Low | 3363 (10%) | 0 (0%) | 44032 (9%) | 4147 (12%) | 4 (6%) | 79488 (16%) | 4889 (15%) | 4 (6%) | 114944 (24%) |
| Medium | 5006 (15%) | 12 (17%) | 44032 (9%) | 5812 (17%) | 16 (23%) | 79488 (16%) | 6524 (20%) | 16 (23%) | 114944 (24%) |

*Tabla 9.9: Recursos hardware utilizados con tamaño de ventana de 32 [297].*

Las aceleraciones No, Low, y Medium requieren los mismos recursos para las tres técnicas de estimación de movimiento, aunque depende del procesador seleccionado.

Tras estos resultados, vamos a comparar los recursos usados frente al rendimiento obtenido. En la Figura 9.16 mostramos los KPPS frente a los *Logic Elements* para cada procesador Nios II y técnica de estimación de movimiento. Además, se presentan resultados para cada tipo de aceleración, y cada tamaño de ventana. Podemos observar que para Nios II /e/s/f, el algoritmo FST logra menos KPPS con la misma cantidad de lógica que TSST o 2DLOG para todas las aceleraciones, salvo en el caso de aceleración High, donde FST logra un mejor rendimiento que cualquier otra aceleración utilizando menos lógica.

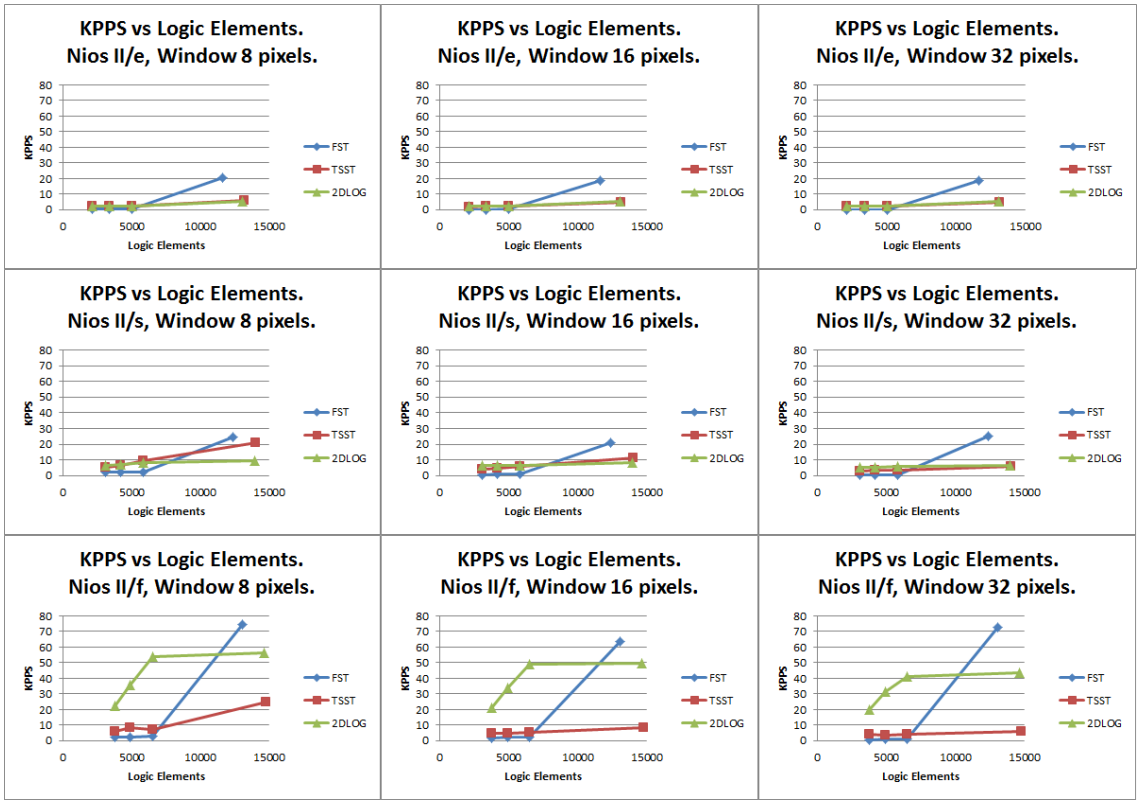


*Figura 9.16: KPPS versus Elementos Lógicos [297].*

En la Figura 9.17 se muestran los KPPS frente a los multiplicadores empotrados para cada procesador Nios II y técnica de estimación de movimiento. Además, se presentan de nuevo resultados para cada tipo de aceleración (No, Low, Medium, y High), y cada tamaño de ventana (8, 16, y 32).

Utilizando el procesador Nios II/e, y aceleraciones No y Low, se obtienen los mismos resultados sin gastar ningún multiplicador. Cuando la aceleración es High, todos los algoritmos utilizan la misma cantidad de multiplicadores, aunque FST logra el mejor rendimiento. Cuando se selecciona el procesador Nios II/s, la FST obtiene los peores resultados con aceleraciones No, Low, o Medium; pero para aceleración High el FST obtiene los mejores resultados con los mismos recursos. Para este procesador TSST obtiene mejores resultados que 2DLOG en todos los casos, excepto en aceleración High con tamaño de ventana 32.

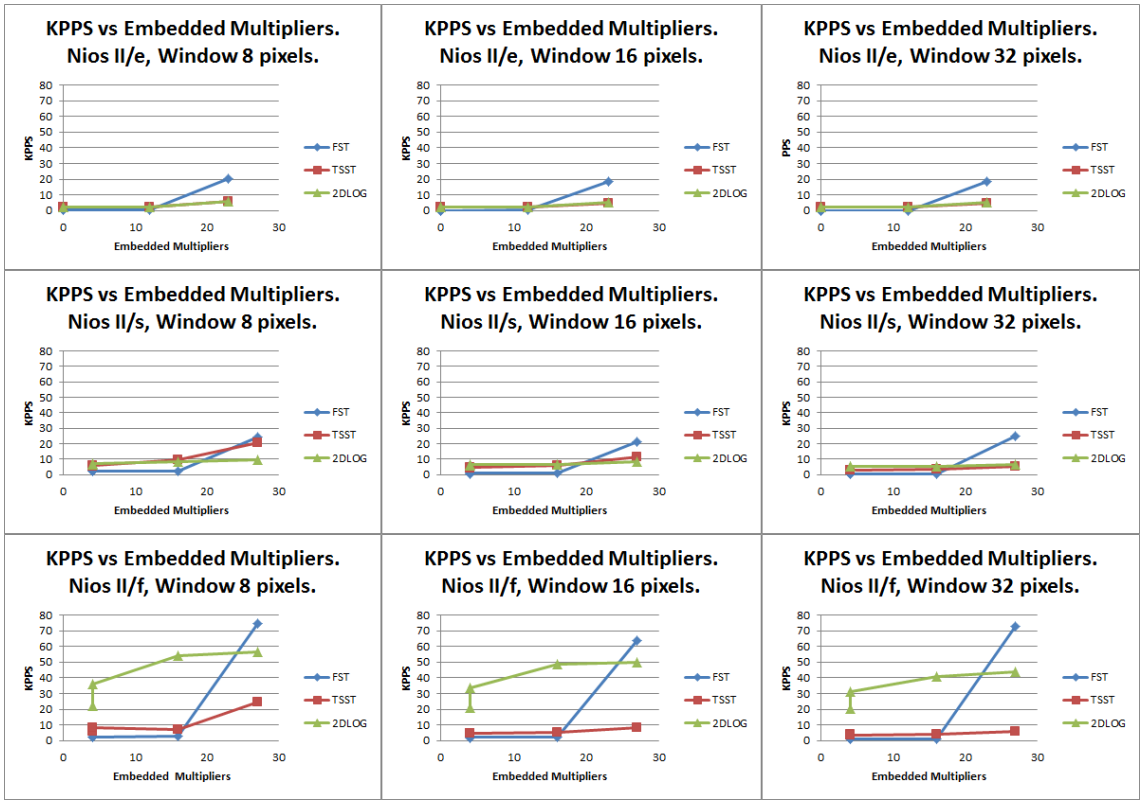


*Figura 9.17: KPPS versus Multiplicadores Empotrados [297].*

Como podemos observar, FST y 2DLOG son los mejores diseños para el procesador Nios II/f, sin embargo 2DLOG obtiene mejores resultados usando aceleraciones del tipo No, Low, o Medium, y finalmente FST, alcanza la mejor cota con aceleración High.

## 9.13. Introducción a las instrucciones personalizadas.

Una instrucción personalizada es una instrucción definida por el usuario y ejecutada en hardware dentro de la estructura del procesador, que realiza las operaciones definidas por el usuario y se añade al repertorio de instrucciones de la arquitectura.

En la Figura 9.18, se muestra cómo funciona la implementación de una instrucción personalizada en paralelo con la ALU del procesador, dentro de la ruta de datos del Nios II. Usando estas instrucciones personalizadas se puede configurar el procesador embebido Nios II para conseguir un mayor rendimiento, entre otros objetivos.

Una vez que la instrucción personalizada se ha agregado a la ruta de datos, podemos instanciarla directamente a través del lenguaje ensamblador del procesador o directamente en nuestro código fuente a través de macros generadas por el Nios II EDS

(*Embedded Design Suite*). Para esto último, se utilizan funciones built-in del compilador GCC (*GNU Compiler Collection*) que pasan directamente del código fuente a la instrucción personalizada a diseñar.

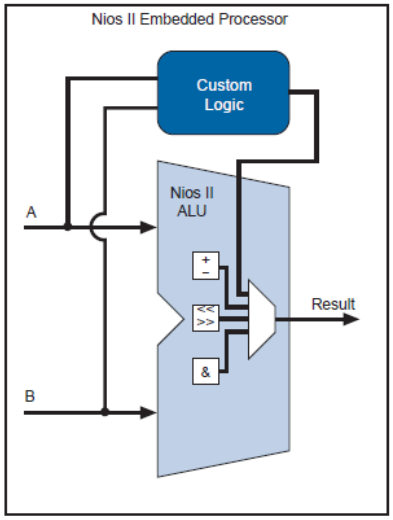


*Figura 9.18: Lógica personalizada conectada a la ALU del Nios II [319].*

### 9.13.1. Tipos de instrucciones personalizadas.

El procesador Nios II permite cinco tipos de instrucciones personalizadas diferentes, "*Combinational*", "*Multi-Cycle*", "*Extended*" y "*Internal Register File*". En la Figura 9.19, se muestra los puertos adicionales de los diferentes tipos de instrucciones personalizadas admitidas con sus entradas y salidas.

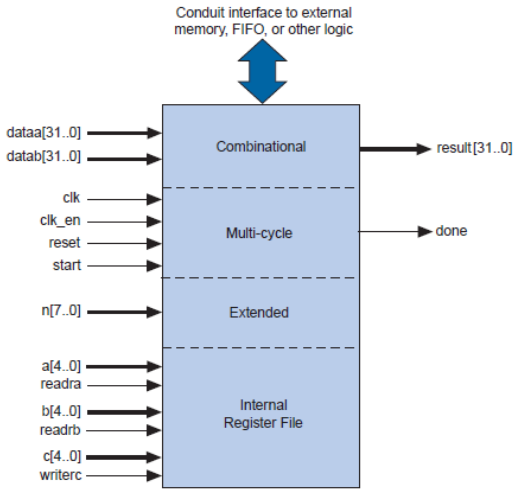


*Figura 9.19: Instrucciones a medida para el Nios II. [319].*

Además de los cuatro tipos de instrucciones personalizadas presentados, hay un quinto, "*External Interface*", que ofrece al diseñador la capacidad de interconectarse con lógica externa a la ruta de datos del procesador Nios II y presentamos aparte.

Una vez abordada una breve introducción sobre las diferentes tipos de instrucciones personalizadas, se procede a su descripción detallada:

- *Combinational* (Combinacional): este tipo de instrucción personalizada proporciona un bloque de lógica personalizada que completa su ejecución en un solo ciclo. En la Figura 9.20 podemos ver las entradas ("dataa" y "datab"), que son opcionales, y su salida ("result"), que será leído en el flanco ascendente del siguiente ciclo de reloj

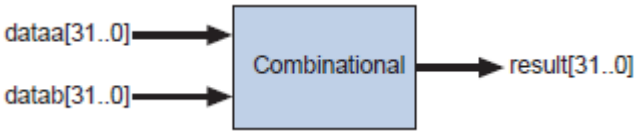


*Figura 9.20: Instrucción personalizada de tipo combinacional [319].*

- *Multi-cycle* (Multiciclo) : este tipo de instrucción personalizada proporciona un bloque de lógica personalizada que necesita más de un ciclo de reloj para ser completado. Este número de ciclos de reloj puede ser fijo o variable dependiendo del diseñador. En la Figura 9.21, podemos ver las entradas ("dataa", "datab", "clk", "clk_en", "reset", y "start") siendo sólo "clk", "clk_en", y "reset" obligatorias, y sus salidas ("result" y "done"), que son opcionales. Si se fija el número de ciclos de reloj, el procesador esperará el número de ciclos especificado para leer "result", pero si el número de ciclos es variable, el procesador espera hasta que se active la señal "done".

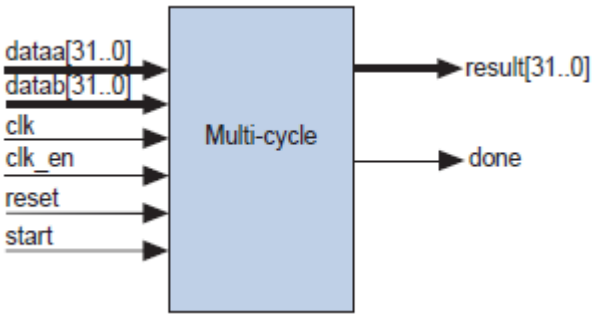


*Figura 9.21: Instrucción personalizada de tipo multiciclo [319].*

- *Extended* (Extendida): este tipo de instrucción proporciona un bloque de lógica personalizada que tiene la capacidad de implementar varias operaciones, hasta 256, a través de un índice que selecciona la operación a ser ejecutada. Estas instrucciones personalizadas también pueden ser de tipo combinacional o multiciclo, gracias al hecho de que las operaciones que contienen, usarán varios valores del índice de selección.

En la Figura 9.22, se presenta un ejemplo de una instrucción personalizada de tipo Extended que tiene tres operaciones diferentes (*bit-swap*, *byte-swap*, y *half-word-swap*). La operación deseada se selecciona a través de la entrada ("n"), y se lleva a cabo sobre el operando de entrada ("dataa"), para finalmente almacenar la solución en la salida ("result").

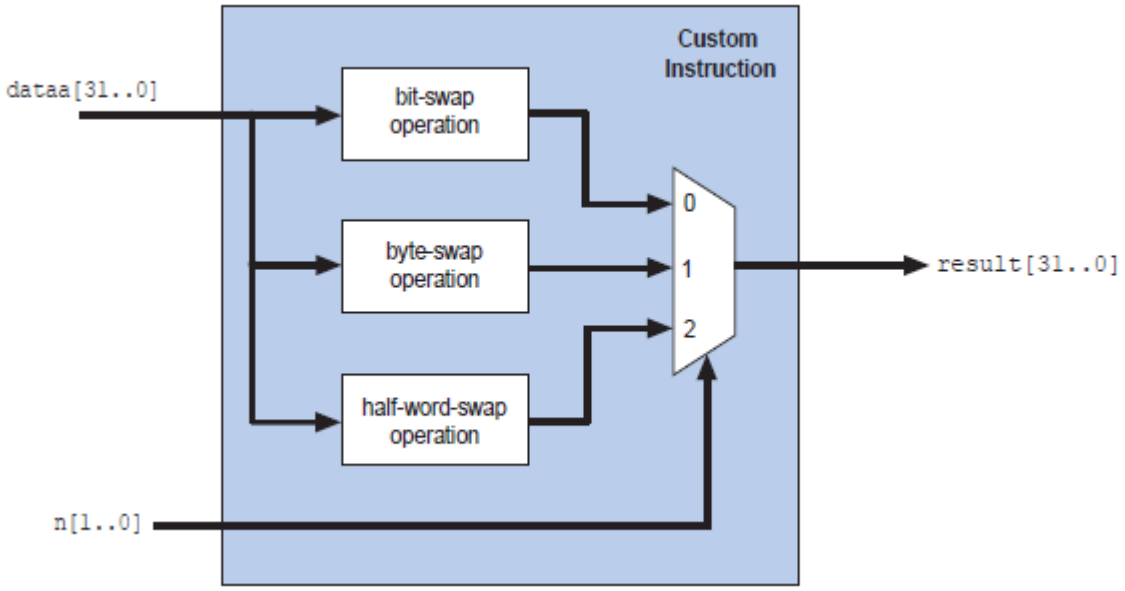


*Figura 9.22: Instrucción personalizada de tipo extendida [319].*

- *Internal register file* (Registro interno): este tipo de instrucción personalizada proporciona un bloque de lógica personalizada que accede a los registros del procesador Nios II, que permite al diseñador leer las entradas y/o escribir la salida desde/a los registros del procesador. Este tipo de instrucción personalizada tiene nuevas entradas ("a", "b", "c", "readra", "readrb", "writerc") añadidas a los puertos de entrada. Con estas entradas ("readra" y "readrb") se decide si la instrucción personalizada carga las entradas de los puertos normales ("data" y "datab") o a través de los registros indexados por los nuevos puertos ("a" y "b"). El mismo proceso se realiza para la salida con la otra

señal (“writerc”) y el otro puerto de entrada (“c”). En la Figura 9.23, se presenta un ejemplo de una instrucción de este tipo que realiza una acumulación de multiplicaciones.

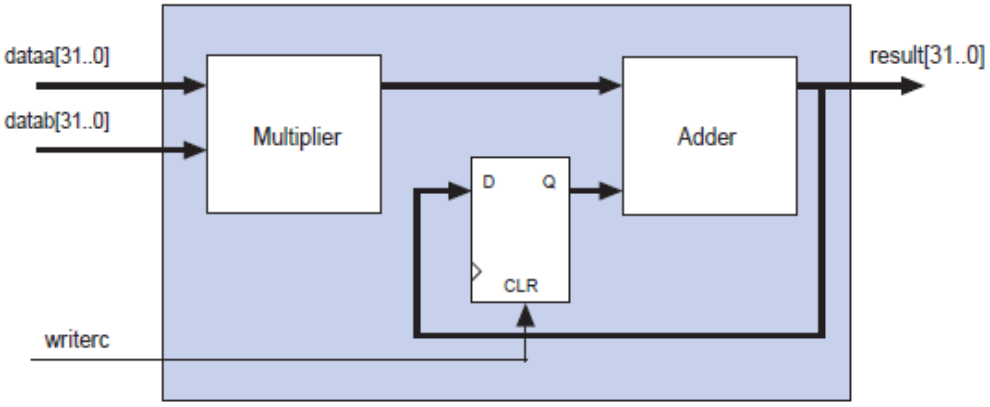


*Figura 9.23: Instrucción personalizada de tipo registro interno [319].*

- *External interface* (Interfaz Externo): este tipo de instrucción personalizada proporciona un bloque de lógica personalizada que puede interactuar con lógica externa a la ruta de datos del procesador Nios II. Este bloque lógico personalizado hereda los otros puertos de instrucciones personalizadas, e incluye también una interfaz definida por el usuario para conectarse a la lógica externa a la ruta de datos del procesador Nios II como se muestra en la Figura 9.24.

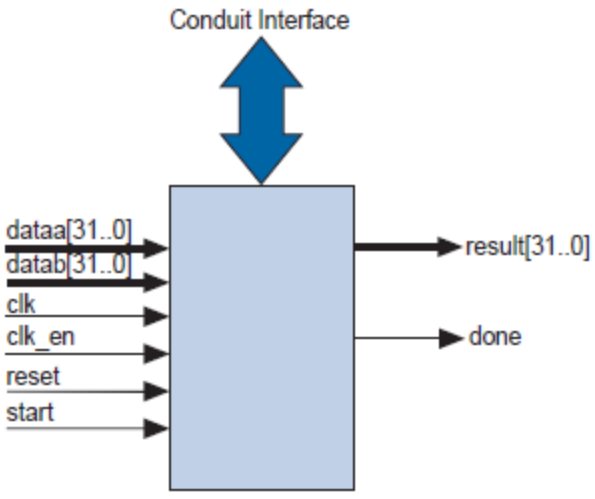


*Figura 9.24: Instrucción personalizada de tipo interfaz externo [319].*

Examinando los análisis de tiempo realizados en el Capítulo 5 podemos sacar conclusiones sobre que parte del código se debe de reemplazar por una instrucción personalizada para obtener un mayor rendimiento. La función GetCost es el primer objetivo debido a que emplea prácticamente todo el tiempo de ejecución.

Después de examinar todos los tipos diferentes de instrucción personalizada, mostrados en el presente apartado, se procede a decidir cuál de ellos podría ser el más adecuado para reemplazar al código fuente de la función GetCost. Nuestro primer enfoque fue claramente una instrucción de tipo combinacional, debido a su velocidad (sólo un ciclo de reloj), su facilidad, y la estructura de la función GetCost.

La instrucción personalizada de multiciclo también se aborda en este trabajo, aunque posterior cronológicamente, logrando una instrucción personalizada más sofisticada que la combinacional. La instrucción personalizada de tipo extendida se descartó debido a que se necesitaba sólo un tipo de operación entre cada par de píxeles (calcular SAD). La instrucción de tipo registro interno emana de la multiciclo que es abordada como se mencionó anteriormente. Por último, la instrucción del tipo interfaz externo ha sido descartada debido al hecho de que no hay necesidad de comunicarse con interfaces externas.

## 9.14. Instrucción personalizada combinacional .

En esta sección, cada resultado obtenido combinando cada procesador Nios II (Nios II / e, Nios II / s, y Nios II / f), todos los tamaños de macrobloque (16, 32, y 64 píxeles), cada algoritmo (FST, 2DLOG, y TSST) y cada tamaño de la ventana de búsqueda (8, 16, y 32 píxeles), se presenta utilizando la instrucción personalizada combinacional diseñada, o sustituyéndose por su código fuente en el software.

En la Figura 9.25 se muestran los resultados descritos para la secuencia de entrada “Foreman” (frames 0 y 1).

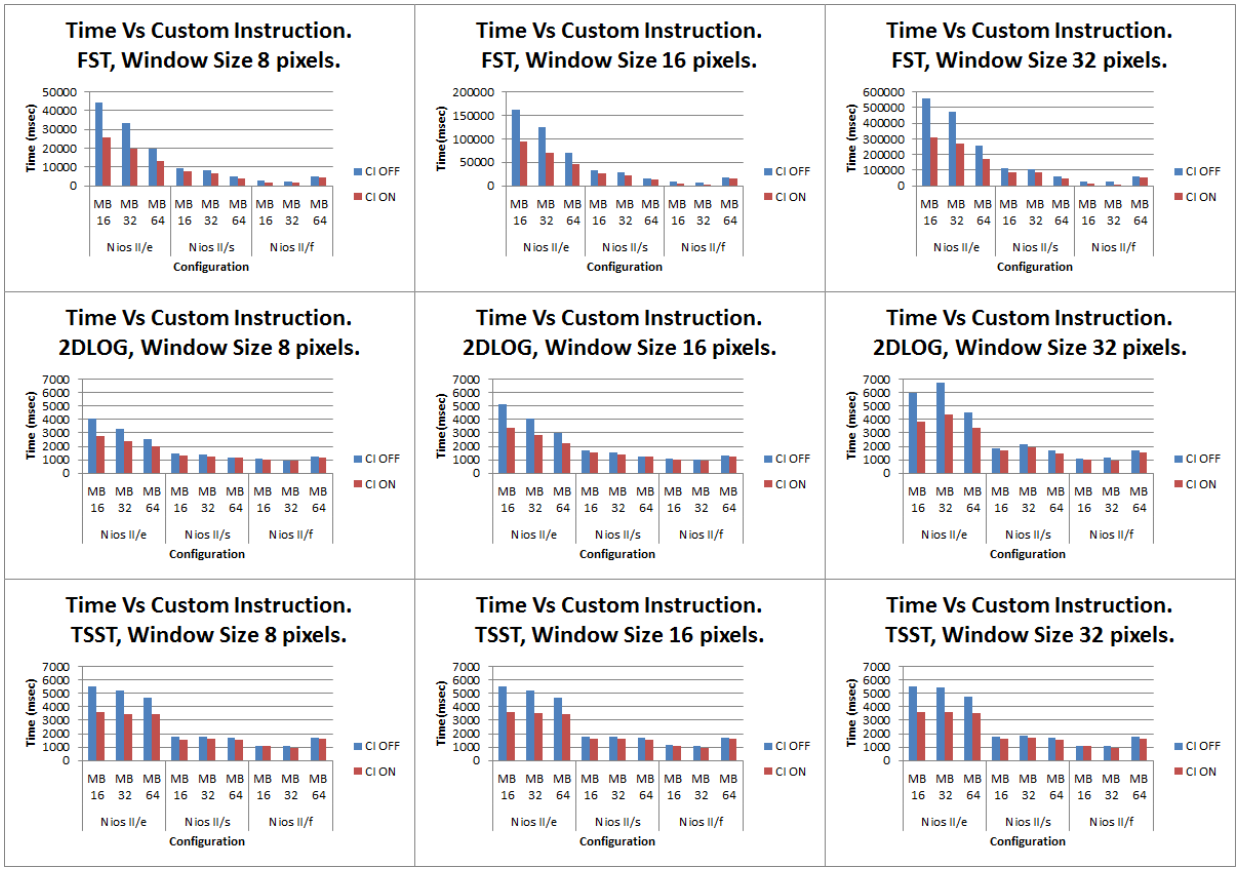


*Figura 9.25: Resultados para la secuencia "Foreman" [320].*

En la Tabla 9.10 se muestra la reducción de tiempo de ejecución para los resultados de la Figura 9.25. De esta manera, es posible ver de un vistazo cuánto tiempo estamos ahorrando al utilizar nuestra instrucción personalizada. En el mejor de los casos se consigue una mejora del 54%, obtenida cuando se ejecuta bajo el procesador Nios II/f el algoritmo FST usando un tamaño de ventana de 32 y un tamaño de macrobloque de 16. Por otro lado, el peor de los casos no obtiene mejora alguna, obtenido al ejecutar bajo el Nios procesador II/f el algoritmo de TSST usando un tamaño de ventana de 32 y un tamaño de macrobloque de 16.

| **Algoritmo / Ventana** | **Procesador** | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | **Nios II/e** | | | **Nios II/s** | | | **Nios II/f** | | |
| | **MB 16** | **MB 32** | **MB 64** | **MB 16** | **MB 32** | **MB 64** | **MB 16** | **MB 32** | **MB 64** |
| FST/8 pixels | 41.72% | 41.88% | 32.23 % | 20.02 % | 16.73% | 15.84% | 39.74% | 36.22% | 9.13% |
| FST/16 pixels | 42.58% | 42.90% | 33.29% | 21.08% | 18.86% | 17.91% | 50.22% | 44.96% | 11.15 % |
| FST/32 pixels | 44.23% | 43.21% | 33.62% | 21.61% | 18.99% | 18.42% | 54.00% | 53.85% | 11.49% |
| 2DLOG/8 pixels | 32.35% | 27.63% | 20.63% | 8.78% | 7.25% | 4.20% | 6.36% | 5.15% | 4.13% |
| 2DLOG/16 pixels | 34.05% | 30.05% | 23.31% | 11.63% | 6.54% | 3.94% | 4.63% | 8.00% | 4.65% |
| 2DLOG/32 pixels | 35.45% | 35.46% | 25.61% | 10.11% | 10.33% | 11.24% | 5.50% | 14.91% | 5.39% |
| TSST/8 pixels | 34.97% | 33.46% | 26.18% | 11.24% | 10.56% | 8.98% | 0.93% | 11.21% | 4.71% |
| TSST/16 pixels | 34.67% | 33.14% | 26.50% | 11.11% | 9.44% | 9.52% | 4.42% | 11.21% | 4.12% |
| TSST/32 pixels | 34.37% | 33.46% | 25.79% | 9.55% | 10.22% | 8.77% | ~0.00% | 11.21 % | 7.39% |

*Tabla 9.10: Tiempo ahorrado para la secuencia "Foreman" [320].*

Ahora, en la Figura 9.26 se muestran los resultados descritos para el segundo estímulo, la secuencia de entrada "Carphone" (frames 0 y 1).

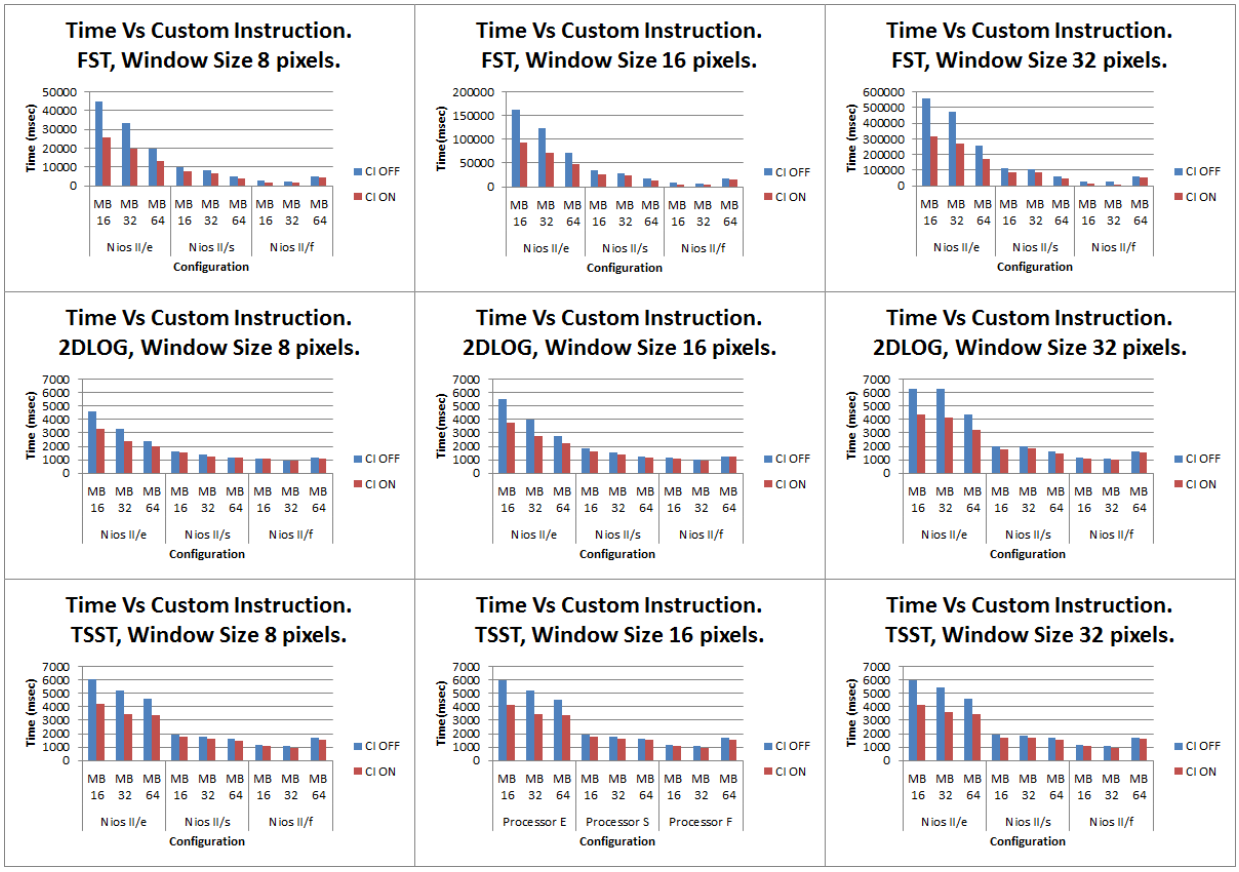


*Figura 9.26: Resultados para la secuencia "Carphone" [320].*

En la Tabla 9.11 se muestra la reducción de tiempo de ejecución para los resultados de la Figura 9.26. De esta manera, es posible ver de un vistazo cuánto tiempo estamos ahorrando al utilizar nuestra instrucción personalizada. En el mejor de los casos se consigue una mejora del 56.1%, obtenida al ejecutar bajo el procesador Nios II/f el algoritmo FST usando un tamaño de ventana de 32 y un tamaño de macrobloque de 16. Por otro lado, el peor de los casos no obtiene mejora alguna, al ejecutar bajo el Nios II/s el algoritmo 2DLOG usando un tamaño de ventana de 8 y un tamaño de macrobloque de 64.

| Algoritmo / Ventana | Procesador | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | Nios II/e | | | Nios II/s | | | Nios II/f | | |
| | **MB 16** | **MB 32** | **MB 64** | **MB 16** | **MB 32** | **MB 64** | **MB 16** | **MB 32** | **MB 64** |
| FST/8 pixels | 42.43% | 41.76% | 31.72% | 19.12% | 17.00% | 15.67% | 40.39% | 35.83% | 9.87% |
| FST/16 pixels | 43.58% | 42.87% | 33.24% | 21.02% | 18.57% | 17.74% | 51.73% | 49.03% | 10.98% |
| FST/32 pixels | 43.91% | 43.17% | 33.69% | 21.44% | 18.99% | 18.48% | 56.01% | 53.73% | 11.53% |
| 2DLOG/8 pixels | 28.76% | 28.53% | 17.92% | 6.79% | 7.91% | ~0.00% | 0.92% | 5.26% | 1.75% |
| 2DLOG/16 pixels | 31.65% | 30.63% | 20.22% | 10.44% | 7.84% | 5.56% | 0.88% | 6.06% | 3.17% |
| 2DLOG/32 pixels | 31.05% | 34.76% | 26.27% | 10.15% | 9.36% | 7.55% | 5.13% | 10.81% | 4.94% |
| TSST/8 pixels | 30.51% | 33.65% | 27.02% | 9.84% | 8.94% | 8.54% | 8.47% | 10.38% | 5.39% |
| TSST/16 pixels | 30.99% | 33.85% | 25.61% | 10.77% | 10.11% | 7.32% | 6.72% | 11.82% | 5.39% |
| TSST/32 pixels | 30.92% | 33.52% | 25.43% | 10.42% | 10.22% | 7.78% | 9.24% | 10.19% | 5.88% |

*Tabla 9.11: Tiempo ahorrado para la secuencia "Carphone" [320].*

## 9.15. Instrucción personalizada multiciclo.

En esta sección abordamos el uso de una instrucción personalizada diseñada como tipo *Multi-cycle*, para reemplazar el código fuente de la función GetCost tal y como ha sido señalado por los diversos análisis realizados en la Sección 5.3.1, en sustitución de la instrucción personalizada de tipo combinacional presentamos previamente.

Como se ha explicado antes, la función GetCost se utiliza para calcular el SAD entre dos macrobloques, uno del fotograma de referencia, y el otro del fotograma actual. Al cambiar el código fuente de la función GetCost por la instrucción personalizada

combinacional, esta última se llama una vez por cada pareja de píxeles, uno de cada macrobloque, acumulando los SAD locales para proporcionar posteriormente el SAD total entre el par de macrobloques necesario.

Sin embargo, cuando sustituimos el código fuente de la función GetCost por la instrucción personalizada multiciclo, esta última es llamada para cada grupo de ocho píxeles, -cuatro de cada macrobloque-, y su principal característica, la arquitectura multiciclo, se utiliza para calcular un SAD acumulado para ese grupo.

De este modo, la ejecución de la instrucción personalizada multiciclo produce pequeños SAD acumulados que posteriormente se suman para lograr el SAD total entre el par de macrobloques requerido.

En la Figura 9.27, se presentan todos los resultados obtenidos para cada combinación posible entre el algoritmo ejecutado (FST, 2DLOG, y TSST), el tamaño del macrobloque seleccionado (16, 32, y 64), y el tamaño de la ventana utilizada (8, 16, y 32), para la secuencia “Foreman” (frames 0 y 1) usando la instrucción personalizada multiciclo.

Los resultados también muestran la instrucción personalizada combinacional y el caso base, que representa el código fuente GetCost traducido directamente a instrucciones del procesador Nios II.

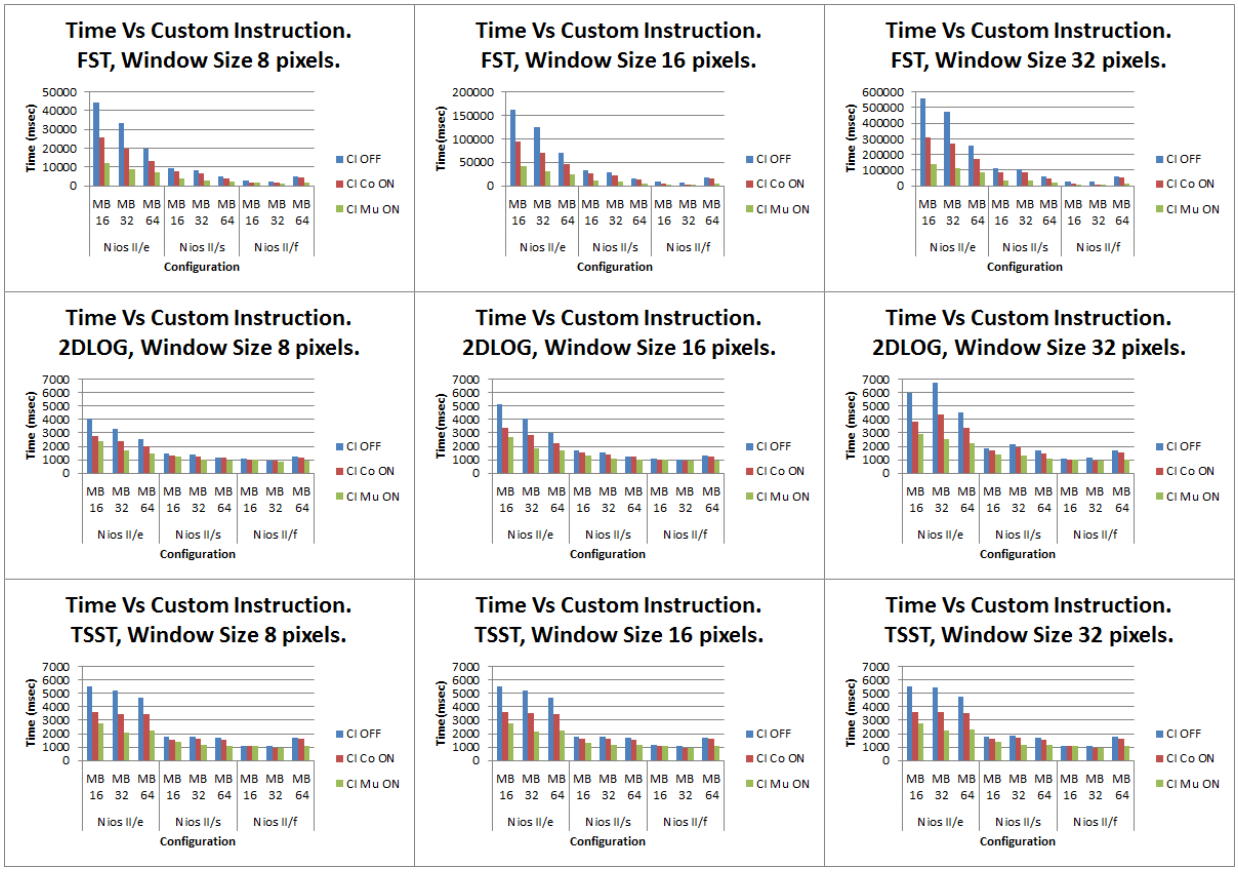


*Figura 9.27: Resultados para la secuencia "Foreman".*

En la Tabla 9.12 se muestra la reducción de tiempo para los resultados de la Figura 9.27. De este modo, es posible analizar gracias a una inspección visual rápida cuanto tiempo estamos ahorrando al utilizar nuestra instrucción personalizada.

El mejor de los casos consigue una reducción del 76.08%, al ejecutar bajo el procesador Nios II/e el algoritmo FST con un tamaño de ventana de 32 y un tamaño de macrobloque de 32.

Por otro lado, el peor caso no obtiene mejora alguna, al ejecutar bajo el Nios II/f el algoritmo TSST usando un tamaño de ventana de 32 y un tamaño de macrobloque de 16.

| **Algorithm / Window size** | **Processor** | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | **Nios II/e** | | | **Nios II/s** | | | **Nios II/f** | | |
| | **MB 16** | **MB 32** | **MB 64** | **MB 16** | **MB 32** | **MB 64** | **MB 16** | **MB 32** | **MB 64** |
| FST/8 pixels | 72.10% | 73.52% | 63.75% | 60.38% | 60.97% | 57.23% | 48.34% | 45.28% | 62.17% |
| FST/16 pixels | 74.52% | 75.53% | 66.39% | 65.94% | 66.39% | 64.96% | 62.81% | 60.03% | 70.14% |
| FST/32 pixels | 75.16% | 76.08% | 67.11% | 67.54% | 67.96% | 67.42% | 67.61% | 65.65% | 72.59% |
| 2DLOG/8 pixels | 40.49% | 49.85% | 40.48% | 16.22% | 24.64% | 20.17% | 8.18% | 9.28% | 23.97% |
| 2DLOG/16 pixels | 46.77% | 54.68% | 42.23% | 22.67% | 28.76% | 22.05% | 4.63% | 10.00% | 25.58% |
| 2DLOG/32 pixels | 50.84% | 62.61% | 51.00% | 25.00% | 39.44% | 34.32% | 6.42% | 17.54% | 37.13% |
| TSST/8 pixels | 50.27% | 59.19% | 52.36% | 23.60% | 35.00% | 34.13% | 0.93% | 14.95% | 37.06% |
| TSST/16 pixels | 49.45% | 58.43% | 52.35% | 25.56% | 33.89% | 32.74% | 6.19% | 13.08% | 37.65% |
| TSST/32 pixels | 48.99% | 59.38% | 51.57% | 23.03% | 37.10% | 33.33% | 0.00% | 14.95% | 38.64% |

*Tabla 9.12: Tiempo ahorrado para la secuencia "Foreman".*

Ahora, en la Figura 9.28 se presentan los resultados para el segundo estímulo, la secuencia "Carphone" (frames 0 y 1).

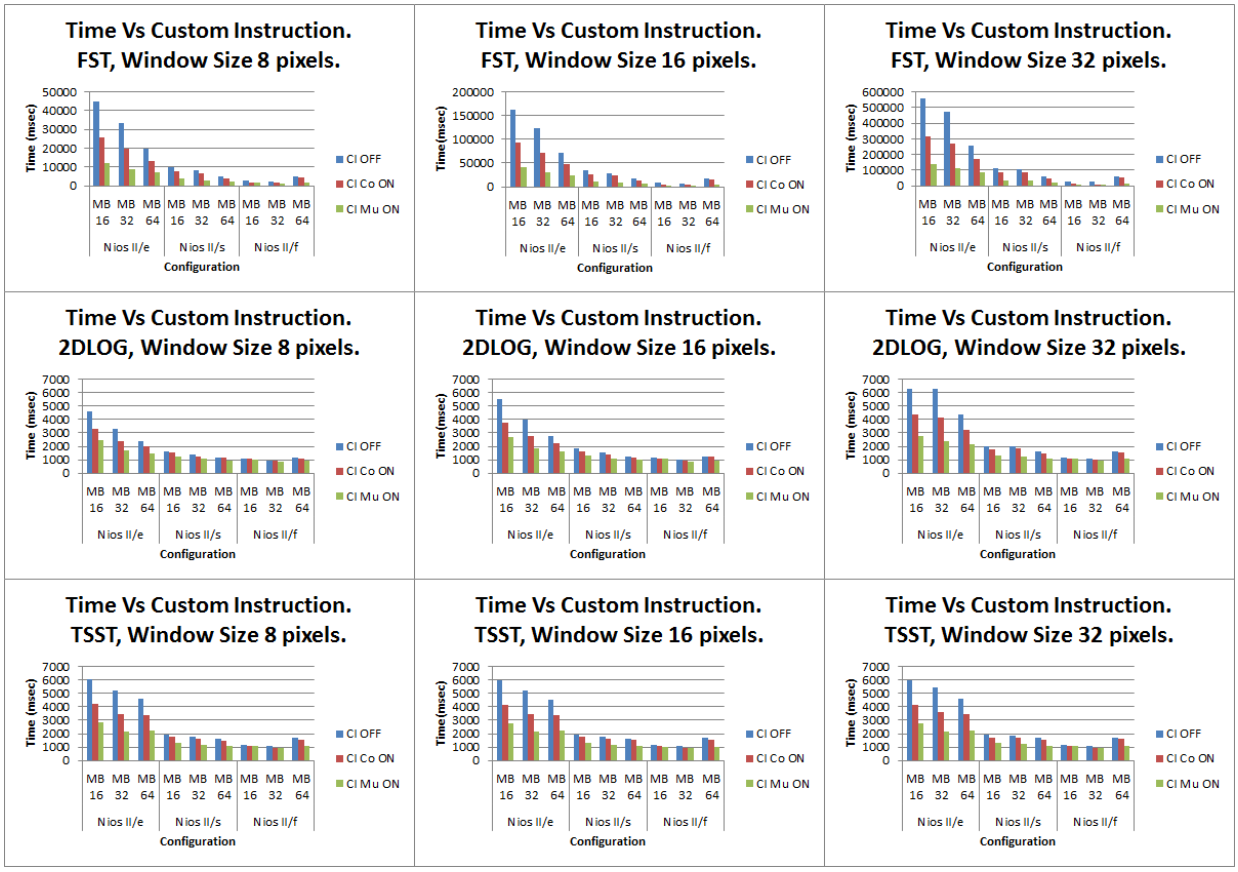


*Figura 9.28: Resultados para la secuencia "Carphone".*

En la Tabla 9.13, se muestra la reducción del tiempo para los resultados de la Figura 9.28. Una vez más, de esta manera, es posible ver de un vistazo cuánto tiempo se ahorra al usar la instrucción personalizada que hemos diseñado.

El mejor de los casos consigue una mejora del 76.07% al ejecutar bajo el procesador Nios II/e el algoritmo FST con una ventana de búsqueda de 32 y un tamaño de macrobloque de 32.

Por otro lado, el peor caso alcanza una mejora del 6,19%, al ejecutar bajo el procesador Nios II/f el algoritmo 2DLOG con una ventana de búsqueda de 16 y un tamaño de macrobloque de 16.

| Algorithm / Window size | Processor | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | Nios II/e | | | Nios II/s | | | Nios II/f | | |
| | **MB 16** | **MB 32** | **MB 64** | **MB 16** | **MB 32** | **MB 64** | **MB 16** | **MB 32** | **MB 64** |
| FST/8 pixels | 72.35% | 73.41% | 63.69% | 60.12% | 60.79% | 56.94% | 49.51% | 45.67% | 61.86% |
| FST/16 pixels | 74.61% | 75.55% | 66.42% | 66.30% | 66.31% | 64.94% | 62.54% | 59.97% | 70.01% |
| FST/32 pixels | 75.16% | 76.07% | 67.15% | 67.80% | 67.95% | 67.42% | 67.78% | 65.67% | 72.60% |
| 2DLOG/8 pixels | 45.97% | 49.25% | 38.75% | 22.22% | 21.58% | 17.24% | 6.42% | 7.37% | 20.18% |
| 2DLOG/16 pixels | 51.54% | 52.66% | 40.79% | 28.57% | 29.41% | 20.63% | 6.19% | 11.11% | 26.19% |
| 2DLOG/32 pixels | 55.57% | 62.06% | 50.69% | 31.47% | 38.42% | 32.08% | 6.84% | 14.41% | 34.57% |
| TSST/8 pixels | 53.40% | 59.04% | 51.63% | 30.05% | 34.64% | 32.32% | 10.17% | 14.15% | 36.53% |
| TSST/16 pixels | 54.27% | 58.46% | 50.99% | 30.77% | 33.71% | 31.71% | 11.76% | 16.36% | 37.13% |
| TSST/32 pixels | 54.12% | 60.07% | 51.52% | 31.25% | 34.95% | 32.93% | 10.92% | 15.74% | 37.06% |

*Tabla 9.13: Tiempo ahorrado para la secuencia "Carphone".*

## 9.16. Tipos de memoria y diseños de sistemas de memoria.

En esta sección se presentan en primer lugar las memorias disponibles en la FPGA Cyclone II de Altera que se van a utilizar en nuestros diseños de los sistemas de memoria. En segundo lugar, se explican los diferentes parámetros de configuración de memoria. Con el uso de estos parámetros de configuración de memoria y la combinación de cada uno de ellos, teniendo en cuenta todas las memorias seleccionadas en la FPGA, presentamos los resultados de rendimiento obtenidos.

### 9.16.1. Tipos de memoria seleccionados.

Aunque en la Sección 4.5.3 hemos presentado todos los tipos de memoria disponible en nuestra FPGA de Altera, sólo hemos utilizados dos de ellos en el proceso de diseño que fueron seleccionados de la siguiente manera.

La memoria Flash se descartó como opción debido a su alta latencia y baja capacidad. Además, en este diseño no es necesaria memoria de tipo no volátil. La memoria SRAM se descartó por su alto coste por MByte, más alto que el de la SDRAM, y en un sistema de bajo coste esto es de primordial importancia. La memoria On-chip se utilizó por ser el tipo de memoria más rápido en el FPGA y también por ser de bajo coste. La memoria SDRAM se utilizó también en el proceso de diseño debido a su bajo coste y a su alta capacidad. Además, necesitábamos otra memoria aparte de la On-chip para almacenar todos los algoritmos y datos manejados en este trabajo.

### 9.16.2. Parámetros de configuración en el proceso de diseño.

A continuación, se describen los diferentes parámetros de configuración en los que se establecen los diferentes tipos de memoria seleccionados. Al combinarlos con cualquier tipo de memoria seleccionado obtenemos todos los diseños de sistema de memoria presentados.

- *Processor reset vector*: indica el módulo de memoria de la FPGA donde se almacena el gestor de arranque y la dirección de reset, es decir, la dirección donde se almacena dicho gestor.

- *Processor exception vector*: indica el módulo de memoria de la FPGA donde se almacena el vector de excepciones y su dirección.

- *Stack* (.stack): indica el módulo de memoria de la FPGA donde se almacena la pila del programa ejecutado. La pila se utiliza normalmente para almacenar datos y variables temporales, además de los parámetros de llamada a la función.

- *Heap* (.heap): indica el módulo de memoria de la FPGA donde se almacena el montículo del programa ejecutado. El montículo se utiliza para almacenar la memoria reservada de forma dinámica.

- *Read/write data* (.rwdata): indica el módulo de memoria de la FPGA donde se almacenan los punteros y las variables globales de lectura/escritura.

- *Read only data* (.rodata): indica el módulo de memoria de la FPGA donde se almacenan las variables globales de solo lectura.

- *Program* (.text): indica el módulo de memoria de la FPGA donde se almacena el programa traducido a código ejecutable

En la Figura 9.29 se presenta un ejemplo de cómo se podrían colocar las secciones previamente presentadas, almacenando todas ellos dentro del mismo módulo de memoria.

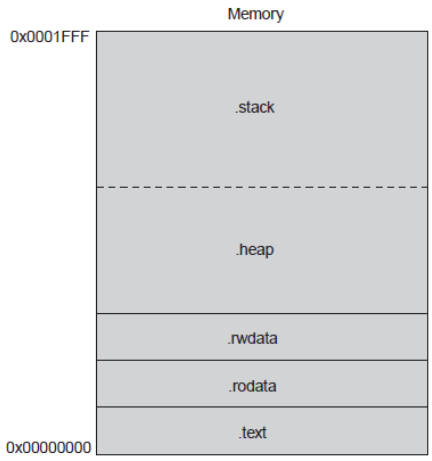


*Figura 9.29: Ejemplo del mapa de memoria [321].*

### *9.16.3. Resultados de los diseños del sistema de memoria.*

En la Tabla 9.14 se muestran todos los diseños del sistema de memoria que proporcionaron un diseño válido, para todas las combinaciones entre los tipos de memoria seleccionados y los parámetros de configuración de nuestra plataforma de pruebas (FPGA Altera DE2-C35, que incorpora un chip Cyclone II EP2C35F672C6N).

Se muestra cada uno de los posibles parámetros de configuración que pueden ser modificadas por el diseñador, así como qué tipo se utilizó de los tipos de memoria seleccionados (On-chip *vs* SDRAM).

| Diseño | Processor reset vector | Processor exception vector | Stack (.stack) | Heap (.heap) | Read/write data (.rwdata) | Read only data (.rodata) | Program (.text) |
|---|---|---|---|---|---|---|---|
| 1 | SDRAM | SDRAM | SDRAM | SDRAM | SDRAM | SDRAM | SDRAM |
| 2 | SDRAM | SDRAM | SDRAM | SDRAM | SDRAM | SDRAM | On-chip |
| 3 | SDRAM | SDRAM | SDRAM | SDRAM | SDRAM | On-chip | On-chip |
| 4 | SDRAM | SDRAM | SDRAM | SDRAM | On-chip | SDRAM | SDRAM |
| 5 | SDRAM | SDRAM | SDRAM | SDRAM | On-chip | On-chip | SDRAM |
| 6 | SDRAM | SDRAM | On-chip | SDRAM | SDRAM | SDRAM | SDRAM |
| 7 | SDRAM | SDRAM | On-chip | SDRAM | On-chip | SDRAM | SDRAM |
| 8 | SDRAM | SDRAM | On-chip | SDRAM | On-chip | On-chip | SDRAM |
| 9 | On-chip | On-chip | SDRAM | SDRAM | SDRAM | SDRAM | SDRAM |
| 10 | On-chip | On-chip | SDRAM | SDRAM | SDRAM | On-chip | SDRAM |
| 11 | On-chip | On-chip | SDRAM | SDRAM | On-chip | SDRAM | SDRAM |
| 12 | On-chip | On-chip | SDRAM | SDRAM | On-chip | On-chip | SDRAM |
| 13 | On-chip | On-chip | On-chip | SDRAM | SDRAM | SDRAM | SDRAM |
| 14 | On-chip | On-chip | On-chip | SDRAM | SDRAM | On-chip | SDRAM |
| 15 | On-chip | On-chip | On-chip | SDRAM | On-chip | SDRAM | SDRAM |
| 16 | On-chip | On-chip | On-chip | SDRAM | On-chip | On-chip | SDRAM |

*Tabla 9.14: Configuración de los diseños de Sistema de memoria [320].*

En la figura 9.30, se presentan en promedio los resultados de la ejecución de las configuraciones anteriores con los estímulos "Foreman" y "Carphone", para cada algoritmo presentado con el procesador Nios II/e. Para ver la mejora de los diseños del sistema de memoria, hemos fijado la ventana de búsqueda a 32 píxeles y el tamaño del macrobloque a 16 píxeles, por ser los parámetros de entrada que hacen que el algoritmo seleccionado tarde más tiempo en ejecutarse, así como el procesador Nios II/e.

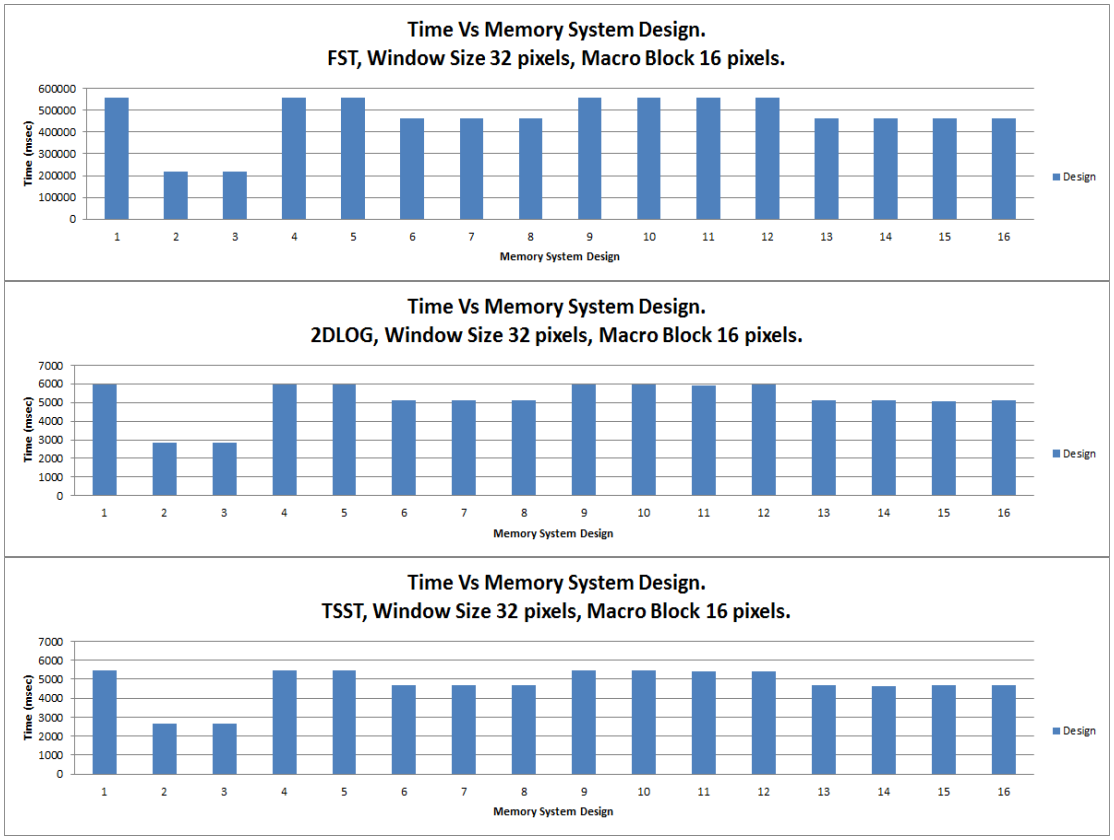


*Figura 9.30: Resultados para "Foreman" y "Carphone", procesador Nios II/e [320].*

Como podemos ver, el grupo formado por las configuraciones de 2 y 3 obtiene el mejor rendimiento, ya que el texto del programa se almacena en la memoria On-chip. El segundo mejor grupo de configuraciones está formado por los diseños del 6 al 8 y del 13 al 16, donde la pila está configurado en la memoria On-chip. El tercer grupo está formado por las configuraciones restantes (1, 4, 5, y 9 a 12), donde la pila y el texto del programa se almacenan en la memoria SDRAM.

El caso base, -diseño número 1-, está construido utilizando la memoria SDRAM para cada parámetro de configuración de los diseños del sistema de memoria.

## 9.17. Instrucción personalizada combinacional añadida al diseño del sistema de memoria.

Una vez presentados estos dos enfoques anteriores, nuestra instrucción personalizada de tipo combinacional y el diseño del sistema de memoria, construimos un sistema que unifique ambas estrategias con el fin de evaluar y mejorar los resultados obtenidos.

Como se ha demostrado antes, sólo hay tres grupos funcionales de configuraciones. Por esta razón, los siguientes resultados se presentan únicamente para las configuraciones del 1 al 8 que representan la gama completa de clases de resultados posibles. Cada configuración presentada fue evaluada incluyendo o no el uso de la instrucción personalizada diseñada, en el lugar de la función GetCost, ejecutando los tres algoritmos presentados en cada procesador Nios II disponible. El tamaño de la ventana de búsqueda se ha fijado a 32 y el tamaño de macrobloque a 16, por ser los valores que tienen tiempos de ejecución más altos. Las Figuras 9.38 y 9.39 muestran los resultados para las secuencias de prueba "Foreman" y "Carphone".

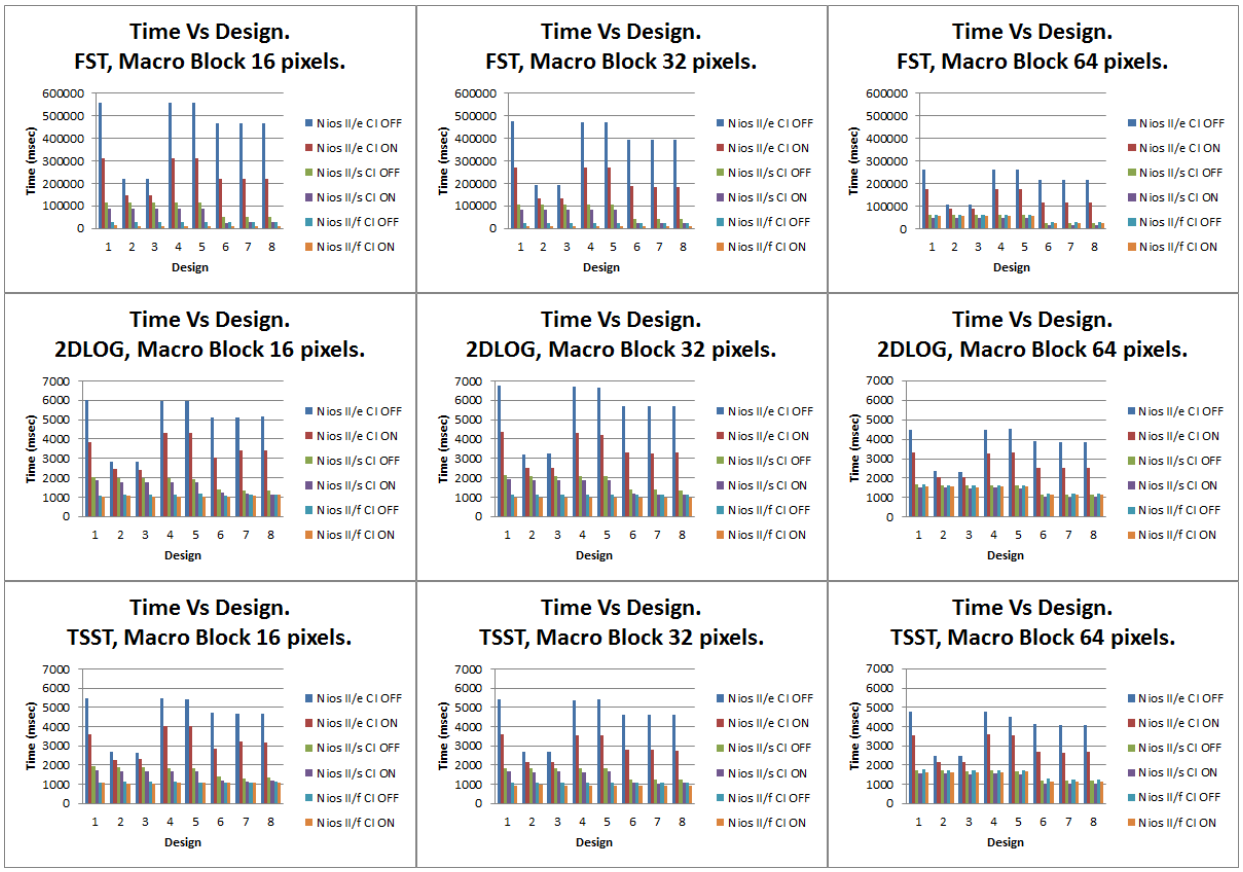


*Figura 9.31: CI combinacional y optimización de memoria. "Foreman" [320].*

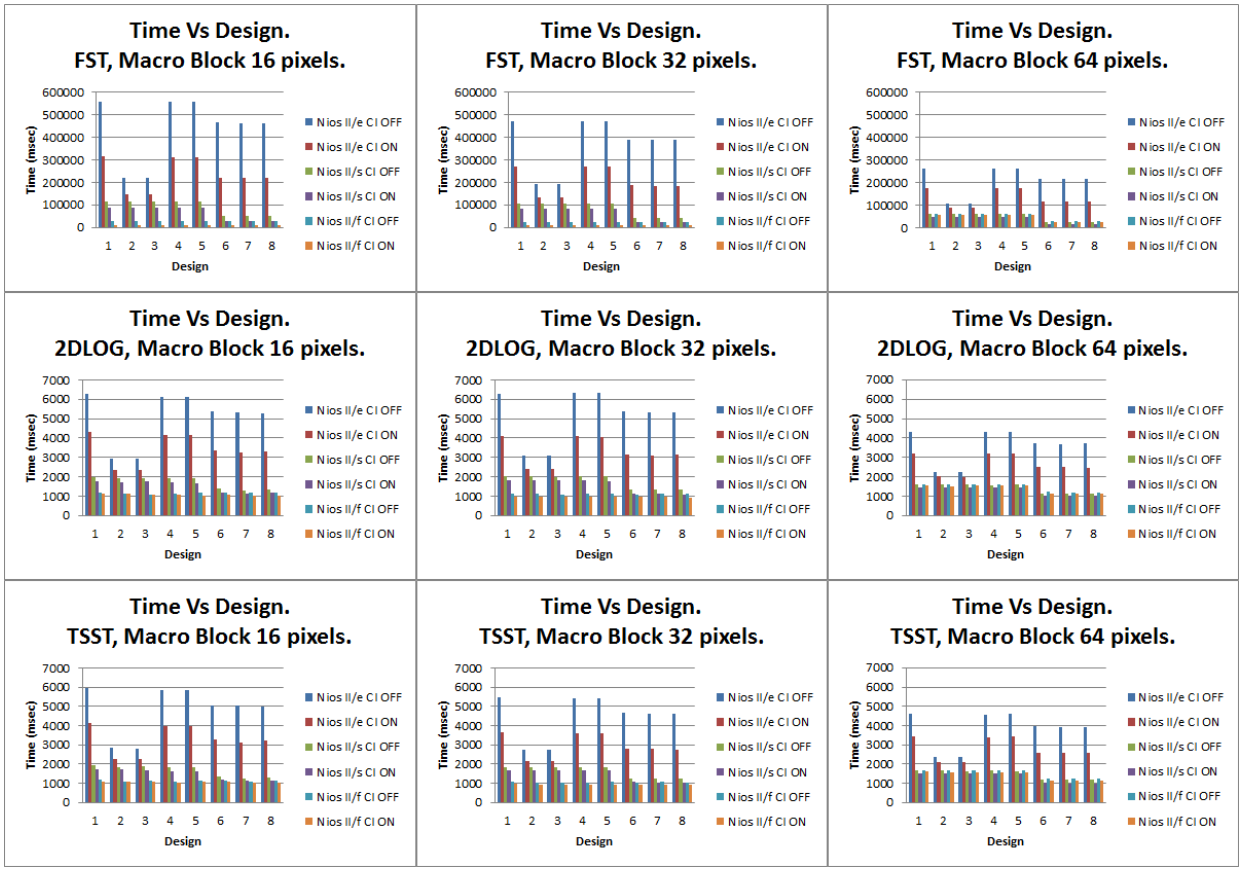


*Figura 9.32: CI combinacional y optimización de memoria. "Carphone" [320].*

## 9.18. Instrucción personalizada multiciclo combinada con diseño del sistema de memoria.

Una vez presentados estos dos enfoques anteriores, nuestra instrucción personalizada de tipo multiciclo y el diseño del sistema de memoria, construimos un sistema unificando ambos con el fin de mejorar los resultados obtenidos.

Las configuraciones 1 al 8, como hemos apuntado antes, representan los tres grupos funcionales de configuraciones. Cada configuración presentada fue probada incluyendo o no el uso de la instrucción personalizada diseñada en lugar del código fuente de la función GetCost, ejecutando los tres algoritmos presentados, en cada procesador.

El tamaño de la ventana de búsqueda se ha fijado a 32 y el tamaño de macrobloque a 16, por ser los valores que tienen tiempos de ejecución más altos.

En la Figura 9.33, se presentan todos los resultados descritos para la secuencia "Foreman" presentando también al mismo tiempo los resultados obtenidos en el caso base (sin CI) y los resultados obtenidos mediante la instrucción combinacional personalizada.

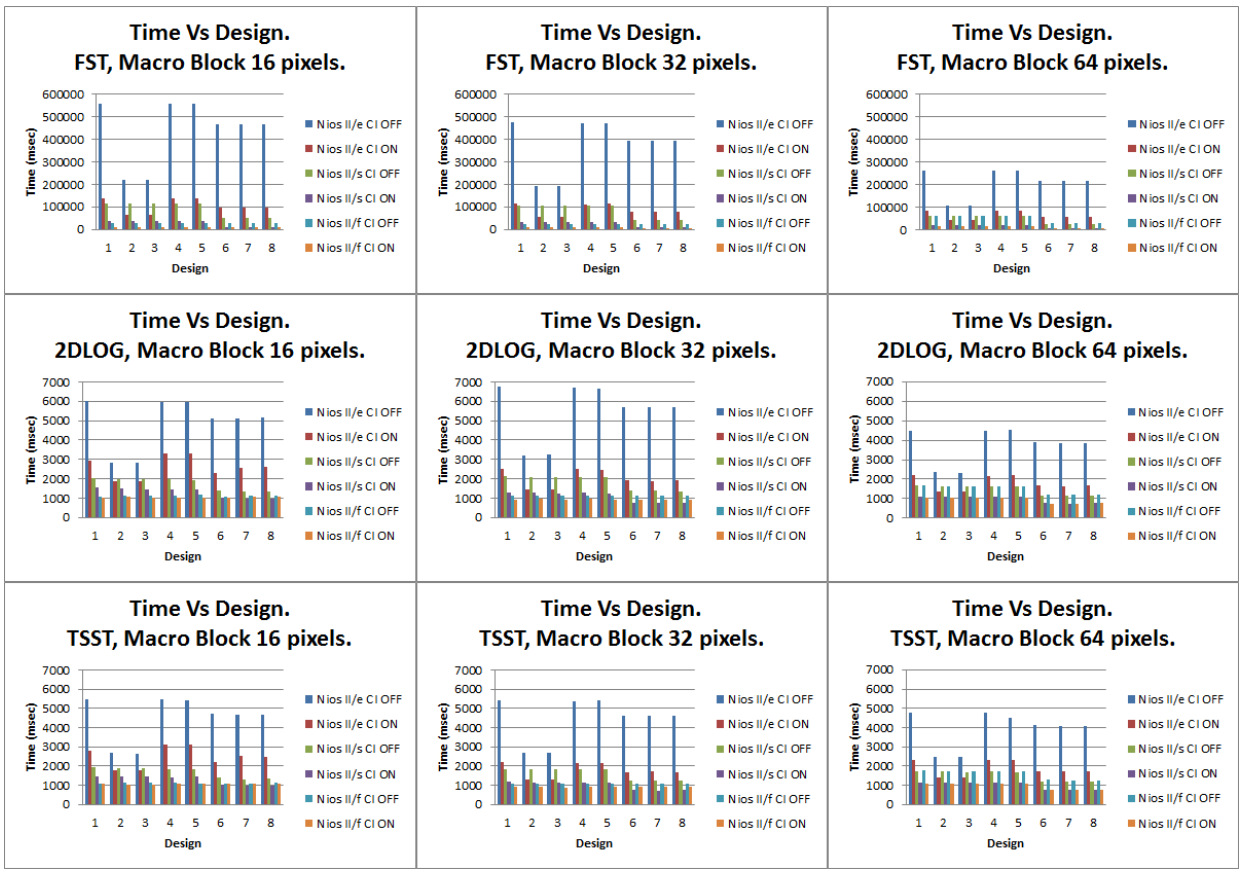


*Figura 9.33:* CI multiciclo y optimización de memoria. *"Foreman".*

Ahora, en la Figura 9.34, se presentan todos los resultados descritos para la secuencia "Carphone" presentando también al mismo tiempo los resultados obtenidos en el caso base (sin CI) y los resultados obtenidos mediante la instrucción combinacional personalizada.

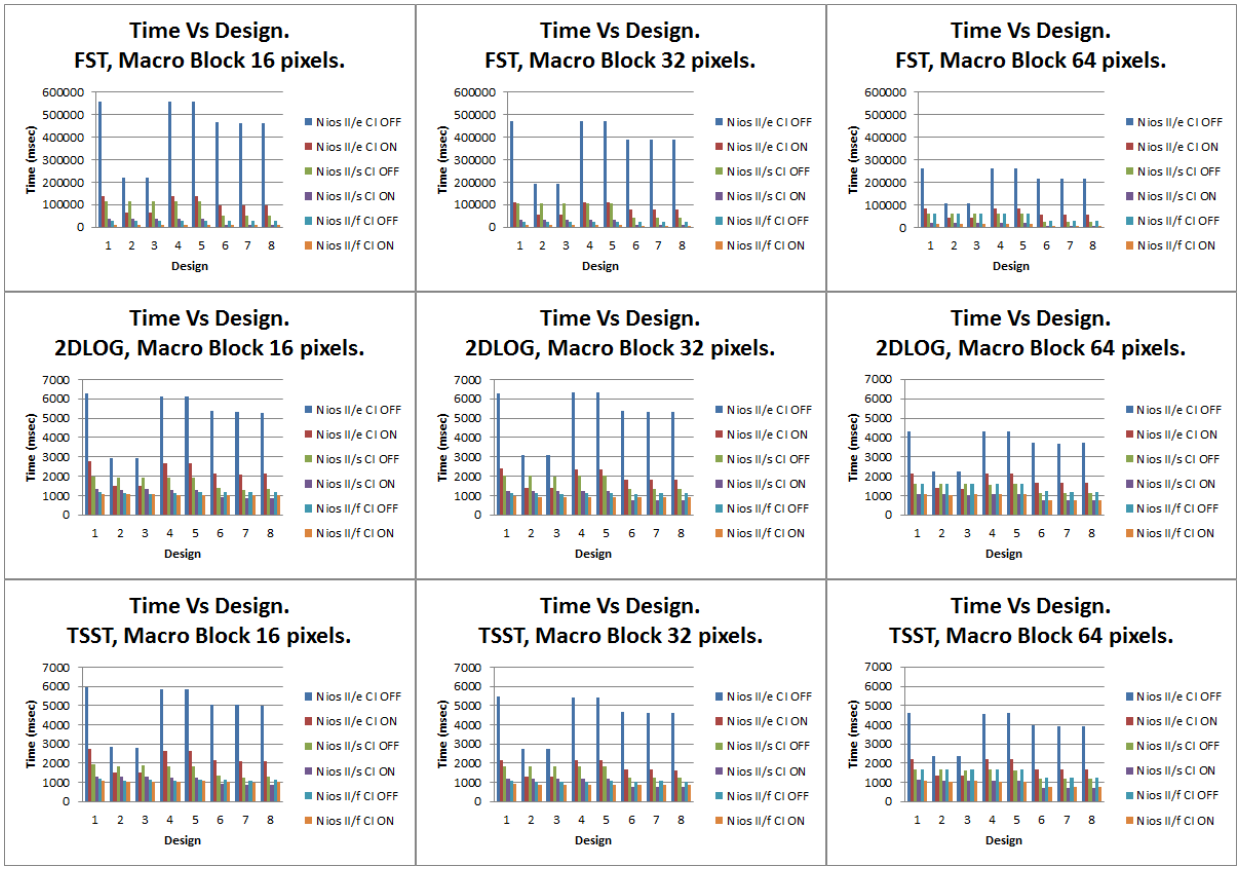


*Figura 9.34:* CI multiciclo y optimización de memoria. *"Carphone"*.

## 9.19. Resultados y análisis crítico.

A continuación, vamos a presentar un análisis de los resultados obtenidos, comparando al mismo tiempo el rendimiento alcanzado en KPPS a través del caso base (sin aceleración y cada parámetro de memoria en la memoria SDRAM), a través de la combinación instrucción personalizada combinacional y el mejor diseño del sistema de memoria en cada caso, tanto para instrucciones monociclo como multiciclo. Figuras 9.43 y 9.44 se muestra el resumen descrito para las secuencias "Foreman" y "Carphone" respectivamente.

En cuanto el rendimiento general del algoritmo FST en cualquier instrucción personalizada, ya sea combinacional o multiciclo obtenemos 6 KPPS en el mejor de los casos, debido al elevado número de operaciones ejecutadas con cada fotograma procesado por esta técnica exhaustiva. Este caso óptimo se alcanza con un tamaño de macrobloque de 64 con el procesador Nios II/s, gracias a la explotación de la caché de instrucciones que este procesador proporciona y el bajo número de iteraciones

necesarias cuando se utiliza este tamaño de macrobloque. Debido a todas las razones explicadas anteriormente en este trabajo, podemos lograr un rendimiento para un pequeño sensor de 50×50 @ 2.5 fps.

En cuanto a la técnica 2DLOG, se consiguen 350 KPPS en el mejor de los casos, debido al hecho de que este algoritmo necesita un menor número de operaciones para procesar una trama que la técnica FST. El mejor de los casos se consigue cuando se utiliza un tamaño de macrobloque de 32 con el procesador Nios II/f, debido a la utilización de las caches de instrucciones y datos proporcionadas por este procesador, cuando se utiliza este tamaño de macrobloque. El rendimiento final obtenido sugiere que, un SoC trabajando con un pequeño sensor de 50×50 @ fps 130 es completamente funcional.

Por último, en relación al algoritmo TSST, se alcanza prácticamente 450 KPPS en el mejor de los casos, debido al hecho de que este algoritmo necesita incluso menos operaciones para procesar una trama que la técnica 2DLOG. El mejor de los casos se consigue cuando se utiliza un tamaño de macrobloque de 32 sobre el procesador Nios II/f, debido a la utilización de las caches de instrucciones y datos proporcionadas por este procesador cuando se utiliza este tamaño de macrobloque. En esta situación se puede construir un SoC que procese 50×50 @ 160 fps y 180 fps, respectivamente para la instrucción combinacional y multiciclo, es decir, una compensación de movimiento en tiempo real para el formato QCIF (alrededor de 19 fps).

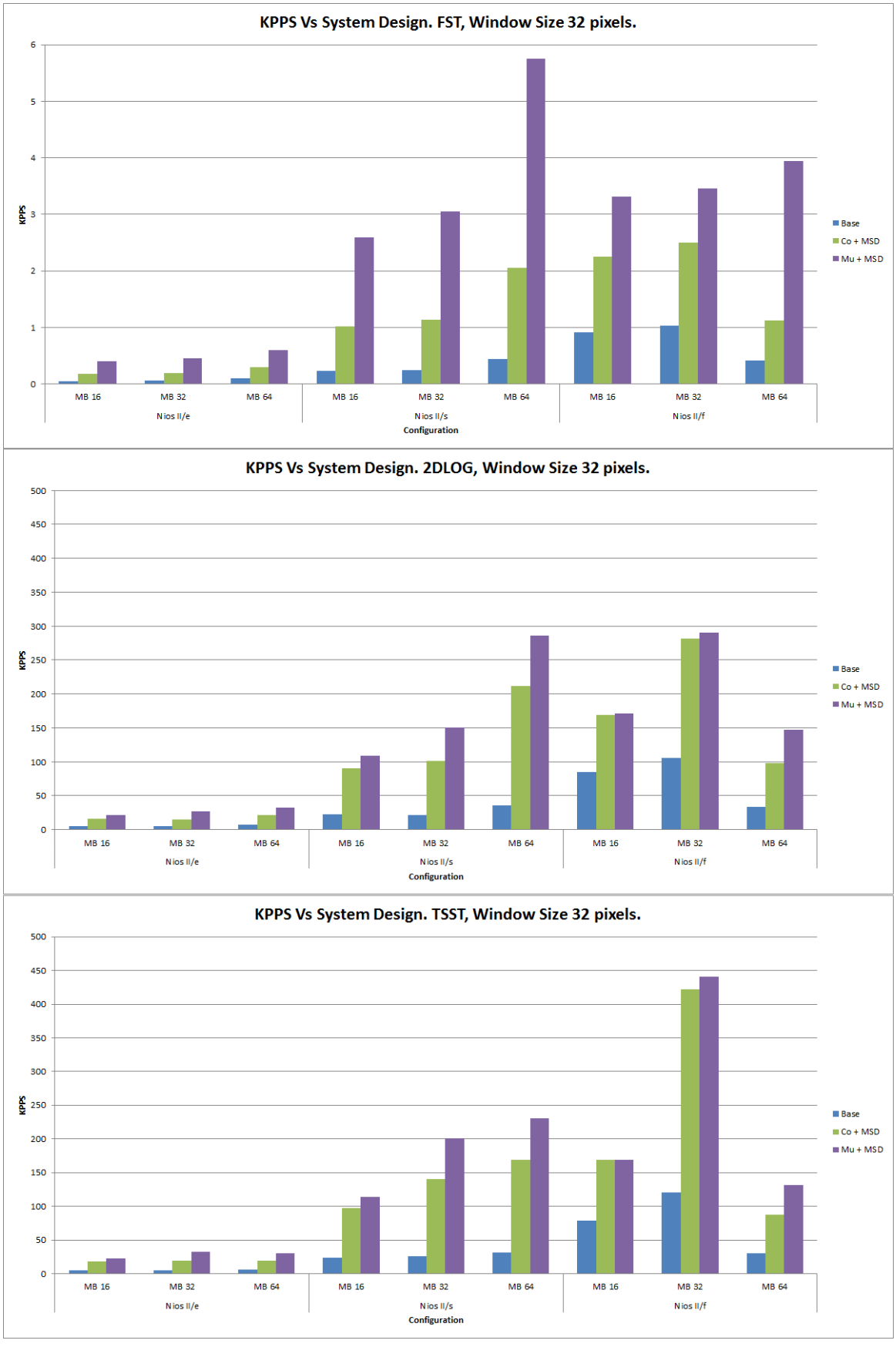


*Figura 9.35: Rendimiento medido en KPPS para "Foreman".*

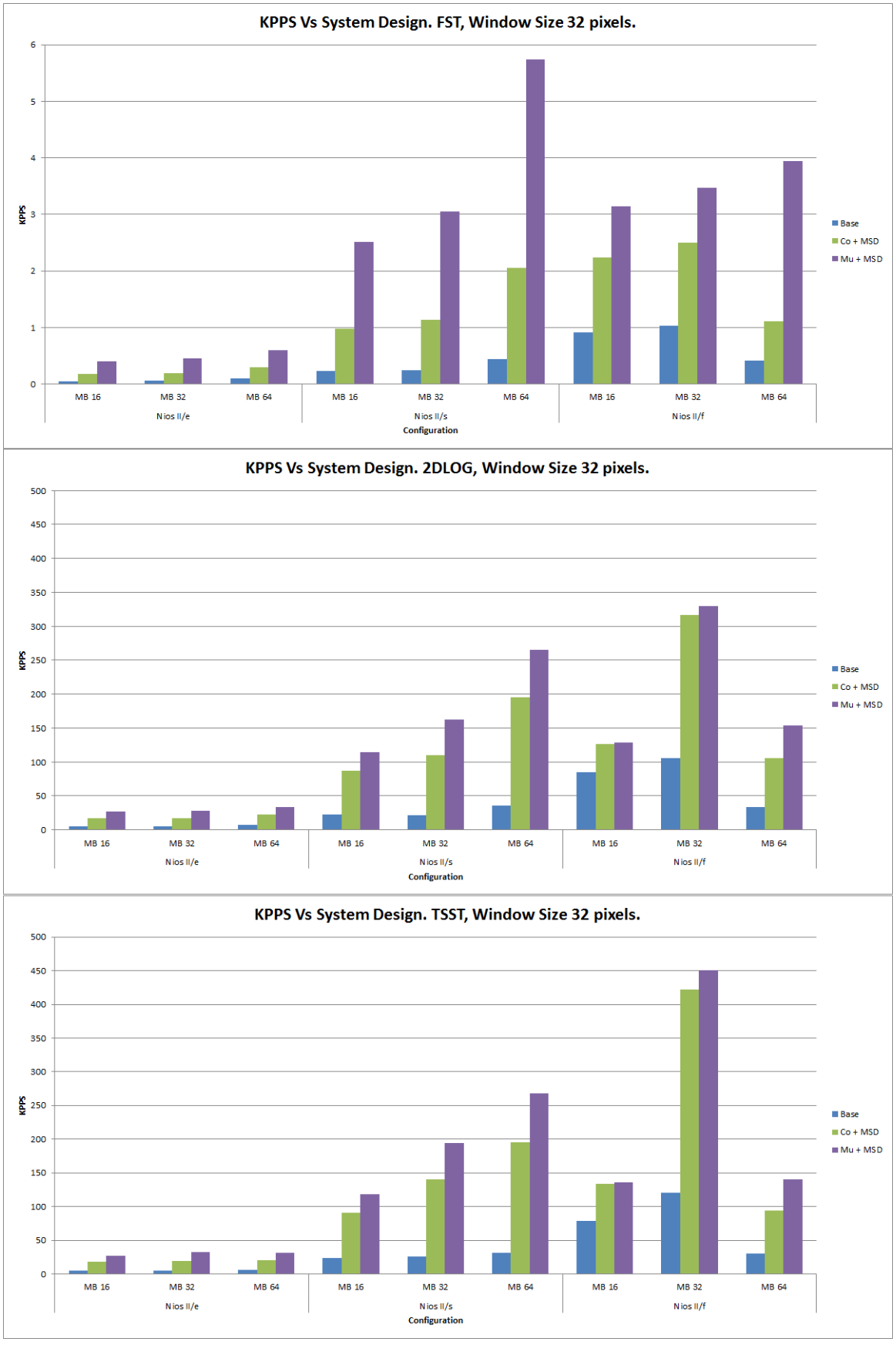


*Figura 9.36: Rendimiento medido en KPPS para "Carphone".*

## 9.20. Conclusiones finales.

En este trabajo de investigación, se han mostrado técnicas para acelerar las rutinas de compensación de movimiento basadas en emparejamiento de patrones como el FST, 2DLOG y TSST. Estos algoritmos reducen el número de bloques a tratar dentro de la ventana de búsqueda mediante patrones de búsqueda estáticos, cuyo objetivo es alcanzar el macrobloque con menos error. Dichas rutinas se ejecutan para sistemas empotrados basados en el microprocesador RISC Nios II y comprenden la parte más costosa y pesada de muchos estándares de codificación de vídeo, tales como H.264, siendo analizadas para conocer su perfil de ejecución.

El primer enfoque que se presenta en este trabajo describe la arquitectura de un sensor de bajo coste en una sistema embebido, utilizando el compilador C2H de Altera con el fin de acelerar las técnicas de estimación de movimiento por correspondencia de bloques. El procesador Nios II permite una gran variedad de complementos, como la SDRAM, UART, SRAM, y las instrucciones personalizadas, pudiéndose incorporar todo en un procesador gracias a la herramienta Altera SOPC. Este enfoque reduce la complejidad de diseño de periféricos hardware, facilitando el desarrollo de un SoC (*System on a Chip*). Este sistema ha sido también caracterizado en términos de precisión con la métrica PSNR, habitual para entornos multimedia. Los resultados obtenidos indican que respecto a la técnica más eficiente y computacionalmente costosa FST, se emplea alrededor del 40% de los LEs y DSPs embebidos, y un 25% de los bloques de memoria RAM. Este sistema es capaz de procesar 72.5 KPPS, equivalente a un SoC que procesa 50×50 @ 29.5 fps.

El segundo enfoque desarrollado, optimiza los algoritmos mencionados anteriormente con la incorporación de instrucciones personalizadas dentro del repertorio de instrucciones del Nios II. Este enfoque se combina con la configuración optimizada de las combinaciones de la memoria SDRAM y la memoria On-chip en el vector de reset, vector de excepciones, pila, montículo, datos de lectura/escritura (.rwdata), datos de solo lectura (.rodata), y el texto de programa (.text) en el diseño. El sistema empotrado final se basa en la combinación eficiente de ambas estrategias.

Con el uso de la instrucción personalizada de tipo combinacional, se alcanzó una mejora del 23% de promedio, y alrededor del 55% en el mejor de los casos, lo que

supone una gran cantidad de ahorro en tiempo de ejecución. Con la optimización de la utilización de los tipos de memoria disponibles en el diseño, se obtuvo una mejora del 61% en el tiempo de ejecución. Y con la combinación de ambas técnicas, se alcanzó una mejora del 70% en promedio, y de un 75% para el caso óptimo comparado respecto al caso secuencial.

Se han diseñado instrucciones personalizadas del tipo multiciclo, mejorando el rendimiento respecto al tipo combinacional y siguiendo la misma metodología mencionada. En este caso ha conseguido un rendimiento promedio del 44% para el conjunto de todos los tamaños de ventana de búsqueda, tamaños de macrobloque, algoritmos, y arquitecturas de procesador utilizadas. Además, la mejora de rendimiento óptima utilizando este diseño es de alrededor un 75% cuando se ejecuta el algoritmo FST bajo el procesador Nios II/e, usando un tamaño de ventana de y macrobloque de 32. Esta mejora en promedio llega al 65% si tenemos en cuenta todas las arquitecturas Nios II con el algoritmo exhaustivo FST. Con la optimización de la utilización de los tipos de memoria disponibles en el diseño, se obtuvo una mejora del 60% en el tiempo de ejecución. Con la combinación de ambas técnicas, se alcanzó una mejora del 80% en promedio, y un 90% para el caso óptimo comparado con el caso base secuencial.

En cuanto al rendimiento óptimo medido en magnitudes multimedia, obtenemos 6 KPPS para la técnica FST, con la arquitectura Nios II/s, 350 KPPS para la técnica 2DLOG y 450 KPPS para la técnica TSST con la arquitectura Nios II/f. Esta tasa permite procesar en tiempo cuasi real QCIF (19 fps) para estos microprocesadores de bajo coste.

El mejor rendimiento hardware se puede considerar siempre usando una estrategia multiciclo combinado con el mejor diseño de sistema de memoria, independientemente del tamaño de macrobloque, la arquitectura escogida, o la secuencia utilizada. En lo que respecta a los recursos hardware necesitados, se puede observar que son bastante livianos para ambas estrategias, al contrario de lo obtenido mediante el uso del C2H.

En conclusión, este trabajo de investigación abre la puerta a la codificación de movimiento para microprocesadores Nios II *soft cores* de coste reducido. Este trabajo presenta contribuciones a distintos campos de investigación como el de Visión por Computador, Codificación Multimedia, y Sistemas Empotrados basados en FPGA.

Una vez que la consistencia de este trabajo ha sido presentada, podemos comentar cuatro líneas principales de investigación que surgen en el desarrollo de nuestros experimentos

## 9.21. Proyección de futuro.

- Nuestras líneas futuras de investigación incluyen planes para integrar un método de disparidad binocular completo (correspondencia estéreo), junto con el sensor de la estimación de movimiento presentado en un sistema integrado para el cálculo de movimiento 3D. Tenemos previsto también ampliar el sistema con un FPGA más moderna y elevada que la usada aquí, y evaluar el sistema completo en un pequeño robot, vehículo autónomo o similar. De esta manera, tendríamos una solución asequible para la aceleración de algoritmos de correspondencia de bloques, manteniendo un equilibrio entre precisión y la eficiencia.

- Adicionalmente, hemos planeado un análisis energético, asociando cada instrucción personalizada diseñada con el rendimiento obtenido, para cada arquitectura específica del microprocesador comentado, cada algoritmo de correspondencia de bloques y, cada tamaño de macrobloque y ventana. Para ello, se aborda el diseño, implementación y validación de una infraestructura hardware/software de análisis de consumo potencia para evaluar las placas de bajo coste utilizadas en este trabajo (familia DE2 de Altera). Esta infraestructura debe ser precisa, reproducible, y flexible en cuanto a la evaluación de por un lado: C2H, estrategias de diseño métodos de instrucción personalizada, otros desarrollos futuros de este trabajo; y por el otro, el impacto de las técnicas y mecanismos para reducir el consumo energético. Se espera conseguir una solución compensada entre eficiencia y consumo de energía para los sistemas integrados de bajo coste basados en procesadores Nios II.

- Se espera también, abordar el MPEG-H parte 2 también conocido como H.265 o simplemente *High Efficiency Video Coding Standart* (HEVC), publicado en su primera versión a principios de 2013. Para esta tarea, se debe utilizar una FPGA con más recursos que la usada en este trabajo. Debido a la complejidad algorítmica involucrada, se considera una alternativa la evaluación de nuevas descripciones de alto nivel como OpenCL para FPGA, junto con los nuevos compiladores CAPS y PGI. Nuevos paradigmas de programación basados en directivas como OpenACC, generan

automáticamente el código OpenCL, y son prometedores para algoritmos intensivos, como los tratados en este trabajo. Mediante este entorno se espera conectar automáticamente una descripción basada en C con el dominio de las puertas lógicas dispuestas en las FPGAs.

- En cuanto a otra línea futura de investigación remarcando además que ya se está desarrollando en la actualidad, enfocaremos la protección de propiedad intelectual para sistemas integrados a través de la ofuscación léxica, siendo un mecanismo muy útil para la protección tanto de los algoritmos de codificación multimedia como de su propio contenido. La motivación, la metodología, los experimentos, y los resultados obtenidos, de esta nueva línea están explicados con detalle en el Apéndice I de la presente memoria, utilizando por un lado VHDL y Verilog, y por otro lado ANSI C para el procesador Nios II.

# References


[1] Wandell, Brian A. Foundations of vision. Sinauer Associates, 1995.

[2] Vicki Bruce, Patrick R. Green, Mark A. Georgeson. Visual perception: physiology, psychology and ecology. 2003.

[3] Zhaoping, Li. Understanding Vision: Theory, Models, and Data. Oxford University Press, 2014.

[4] Anderson, Andre J. et. Al. Of words, eyes and brains: Correlating image-based distributional semantic models with neural representations of concepts. Proceedings of EMNLP 2013, 2013.

[5] Read, Jenny CA. The place of human psychophysics in modern neuroscience. 2014.

[6] Wiegand, T., Sullivan, G. J., Bjontegaard, G., Luthra, A. Overview of the H. 264/AVC video coding standard. Circuits and systems for Video Technology, IEEE Transactions on, 13 (7), pp. 560-576. 2003.

[7] Richardson, I. E. H. 264 and MPEG-4 video compression: video coding for next generation multimedia. John Wiley & Sons, 2004.

[8] R Szeliski. Computer vision: algorithms and applications. Springer, 2010.

[9] [online]. NVIDIA corporation. GPU computing definition. http://www.nvidia.com/object/what-is-gpu-computing.html. (Last accessed February 2014).

[10] [online]. NVIDIA corporation. GPU (Graphic Processing Unit). http://www.nvidia.com/object/gpu.html. (Last accessed January 2014).

[11] Jaynes, Christopher Crossroads. Computer vision and artificial intelligence, 3 (1), pp. 7–10. 09/1996, ISSN 1528-4972.

[12] Machine Learning in Computer Vision. Computational Imaging and Vision, 29, p. 252. 01/2005, ISBN 9781402032745.

[13] Olver, Peter J and Tannenbaum. Mathematical methods in computer vision. The IMA volumes in mathematics and its applications, 10, pp. 133-153. Allen, 2003, ISBN 9780387004976.

[14] Simon Webb. Understanding robotic vision systems. PACE, 11/2012, ISSN 1329-6221.

[15] Granlund, Gösta H and Knutsson. Signal processing for computer vision, 12, p. 437. Hans, 1995, ISBN 9780792395300.

[16] Brand, M; Cooper, P; Birnbaum, L. Seeing physics, or: physics is for prediction. Proceedings of the Workshop on Physics-Based Modeling in Computer Vision, p. 144. 1995, ISBN 9780818670213.

[17] Ranchordas, Alpesh Kumar; Pereira, Joao Madeiras; Araujo, Helder J; Tavares, Joao Manuel. Computer Vision, Imaging and Computer Graphics : Theory and Applications.R.S. Communications in Computer and Information Science, 68, p. 375. 01/2010, ISBN 9783642118395.

[18] Reichardt W. Autocorrelation, a Principle for the Evaluation of Sensory Information by the Central Nervous System. Sensor and Communication, pp. 303-317. 1961.

[19] Hubel. D. H., Wiesel T. N. Receptive Fields and Functional Architecture in Two Non-Striate Visual Areas (18 and 19). of the Cat. Journal of Neurophysiology, 28, pp. 229-289. 1965.

[20] Hubel. D. H., Wiesel T. N. Receptive fields and functional architecture of monkey striate cortex. Journal of Physiology, 195, pp. 215-243. 1968.

[21] Bolduc M., Levine M. D. A Real-Time Foveated Sensor with Overlapping Receptive Fields. Real-Time Imaging, 3, pp. 195-212. 1997.

[22] Bayley D. Design for Embedded Image Processing on FPGAs. Chapter IV Languages. Wiley-IEEE Press eBook Chapters, pp. 73-78. 2011.

[23] [online]. Altera Corporation. APEX II Programmable Logic Device Family Data Sheet. http://www.altera.com/literature/ds/archives/ds_ap2.pdf. (Last accessed February 2014).

[24] Cyclone II Handbook. Altera, February 2008.

[25] Quartus II Handbook. Altera, July 2010.

[26] Gibson, J. J. Perception of the Visual World. Houghton Mifflin, Boston, 1950.

[27] Marr, D. Vision. Edit. W.H.Freeman, 1982.

[28] Verri A., Poggio T. Motion Field and Optical Flow: Qualitative Properties. IEEE Transactions on Pattern Analysis and Machine Intelligence, 11, pp. 490-498. 1989.

[29] Nakayama K. Biological Image Motion Processing: A Review. Vision Research, 25, pp. 625-660. 1985.

[30] Mitiche A. Experiments in Computing Optical Flow with the Gradient-Based. Multiconstraint Method, Pattern Recognition, 20 (2), pp. 173-179. 1987.

[31] Mitiche A., Bouthemy P. Computation and Analysis of Image Motion: A Synopsis of Current Problems and Methods. International Journal of Computer Vision, 19 (1), pp. 29-55. 1996.

[32] J.M. Giménez-Amaya. Anatomía funcional de la corteza cerebral implicada en los procesos visuales. Rev. Neurol, 30 (7), pp. 656–662. 2000.

[33] Wallach H. On Perceived Identity: 1. The Direction of Motion of Straight Lines. In On Perception. Quadrangle, New York, 1976.

[34] Horn B. K. P., Schunck B. G. Determining Optical Flow. Artificial Intelligence, 17, pp. 185-203. 1981.

[35] Adelson E. H., Bergen J. R.. Spatiotemporal Energy Models for the Perception of Motion. Journal of the Optical Society of America A, 2 (2), pp. 284-299. 1985.

[36] Ong E. P., Spann M. Robust Optical Flow Computation Based on Least-Median-of-Squares Regression. International Journal of Computer Vision, 31 (1), pp. 51-82. 1999.

[37] H. Oh and H. Lee. Block-Matching algorithm based on an adaptive reduction of the search area for motion estimation. Real-Time Imaging, 6 (5), 407–414. October 2000.

[38] Oh H., Lee H. Block-matching algorithm based on an adaptive reduction of the search area for motion estimation, Real-Time Imaging, 6, pp.407-414. 2000.

[39] Baker, S., Matthews, I. Lucas-Kanade 20 Years On: A Unifying Framework. International Journal of Computer Vision,56 (3), pp. 221-255. 2004.

[40] Lucas B. D., Kanade T. An Iterative Image Registration Technique with an Application to Stereo Vision. Proceedings of the Seventh International Joint Conference on Artificial Intelligence, pp. 674-679. 1981.

[41] Huang C. and Chen, Y. Motion Estimation Method Using a 3D Steerable Filter. Image and Vision Computing, 13 (1), pp.21-32. 1995.

[42] [online]. Nyquist–Shannon sampling theorem. http://en.wikipedia.org/wiki/Nyquist–Shannon_sampling_theorem. (Last accessed March 2014).

[43] Yacoob. Y. and Davis. L. S. Temporal Multi-scale Models for Flow and Acceleration. International Journal of Computer Vision, 32 (2), pp.1-17. 1999.

[44] Christmas W. J. Spatial Filtering Requirements for Gradient-Base Optical Flow Measurements. Proceedings of the British Machine Vision Conference, pp.185-194. 1998.

[45] Zhang J. Z., Jonathan Wu. Q. M. A Pyramid Approach to Motion Tracking. Real-Time Imaging, 7, pp. 529-544. 2001.

[46] Albright T. D. Cortical Processing of Visual Motion. Miles & Wallman, Visual Motion and its Role in the Stabilisation of Gaze, pp. 177-201. 1993.

[47] Reichardt, W. Autocorrelation, a principle for the evaluation of sensory information by the central nervous system. Sensory communication (MIT Press), pp. 303–317. Rosenblith, 1961.

[48] [online]. Electrophysiology. http://www.neuro.mpg.de/30089/modelfly. (Last accessed February 2014).

[49] Beare R., Bouzerdoum A. Biologically inspired Local Motion Detector Architecture. Journal of the Optical Society of America A, 16 (9), pp. 2059-2068. 1999.

[50] Zanker J. M. On the Elementary Mechanism Underlying Secondary Motion Processing. Phil. Transactions of the Royal Society of London, B, 351, pp. 1725-1736. 1996.

[51] Arias-Estrada M., Tremblay M., Poussart D. A Focal Plane Architecture for Motion Computation. Real-Time Imaging, 2, pp. 351-360. 1996.

[52] Rosin P.L. Thresholding for Change Detection. Proceedings of the International Conference of Computer Vision, pp. 274-279. 1998.

[53] Pajares, G. De la Cruz, Jesús. Visión por Computador (imágenes digitales y aplicaciones). 2003, ISBN 84-7897-472-5. RA-MA.

[54] Ozalevli, E. Higgins, C.M. Reconfigurable biologically inspired visual motion systems using modular neuromorphic VLSI chips. Circuits and Systems I: Regular Papers, IEEE Transactions on , 52 (1), pp.79,92. 2005, doi: 10.1109/TCSI.2004.838307.

[55] Anandan P., Bergen J. R., Hanna K. J., Hingorani R. Hierarchical Model-based Motion Estimation. Motion Analysis and Image Sequence Processing. Kluwer Academic Publishers, M.I.Sean and R.L.Lagendijk editors, 1993.

[56] Accame M., De Natale F. G. B., Giusto D. High Performance Hierachical Block-based Motion Estimation for Real-Time Video Coding. Real-Time Imaging, 4, pp. 67-79. 1998.

[57] Defaux F., Moscheni F. Motion Estimation Techniques for Digital TV: A Review and a New Contribution. Proceedings of the IEEE, 83 (6), pp. 858-876. 1995.

[58] Edward H Adelson and James R Bergen. The plenoptic function and the elements of early vision. Computational models of visual processing, 1 (2). 1991.

[59] E.H. Adelson and J.R. Bergen. Spatiotemporal Energy Models for the Perception of Motion. Journal of the Optical Society of America A, 2 (2), pp. 284–299. 1985.

[60] Weber K., Venkatesh S., Kieronska D. Insect Based Navigation and its Application to the Autonomous Control of Mobile Robots. Proceedings of the International Conference on Automation, Robotics and Computer Vision. 1994.

[61] Fleet D. J., Jepson A. D. Hierarchical Construction of Orientation and Velocity Selective Filters. IEEE Transactions on Pattern Analysis and Machine Intelligence, 11 (3), pp. 315-325. 1989.

[62] Van Hateren J. H., Ruderman D. L. Independent Component Analysis of Natural Image Sequences Yield Spatio-Temporal Filters Similar to Simple Cells in Primary Visual Cortex. Proceedings of the Royal Society of London, B, 265, pp. 2315-2320. 1998.

[63] Berthold KP Horn. Determining lightness from an image. Computer Graphics and Image Processing, 3 (4), 277–299. 1974.

[64] B.K.P. Horn and B.G. Schunck. Determining Optical Flow. Artificial Intelligence, 17, 185–203. 1981.

[65] J.L Barron, D.J. Fleet, and S.S. Beauchemin. Performance of optical flow techniques. INTERNATIONAL JOURNAL OF COMPUTER VISION, 12, pp. 43–77. 1994.

[66] Ben Galvin, Brendan McCane, Kevin Novins, David Mason, and Steven Mills. Recovering motion fields: An evaluation of eight optical flow algorithms. British machine vision conference, 1, pp. 195–204. 1998.

[67] B. McCane, K. Novins, D. Crannitch, and B. Galvin. On benchmarking optical flow. Computer Vision and Image Understanding, 84 (1), pp. 126–143. October 2001.

[68] Sobey P. and Srinivasan M. V. Measurement of Optical Flow by a Generalized Gradient Scheme. Journal of the Optical Society of America A, 8 (9), pp. 1488-1498. 1991.

[69] Arnspang J. Motion Constraint Equations Based on Constant Image Irradiance. Image and Vision Computing, 11 (9), pp. 577-587. 1993.

[70] Ghosal S., Mehrotra R. Robust Optical Flow Estimation Using Semi-Invariant Local Features. Pattern Recognition, 30 (2), pp. 229-237. 1997.

[71] S. Baker and I. Matthews. Lucas-Kanade 20 Years On: A Unifying Framework. International Journal of Computer Vision, 56 (3), pp. 221–255. 2004.

[72] B.D. Lucas and T. Kanade. An Iterative Image Registration Technique with an Application to Stereo Vision (IJCAI). Proceedings of the Seventh International Joint Conference on Artificial Intelligence (IJCAI), pp. 674–679. April 1981.

[73] Uras S., Girosi F., Verri A., and Torre V. A Computational Approach to Motion Perception. Biological Cybernetics, 60, pp. 79-87. 1998

[74] Simoncelli E. P., Adelson E. H., Heeger D. J. Probability Distributions of Optical Flow. Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition. 1991.

[75] Simoncelli E. P., Heeger D. J. A Model of Neuronal Responses in Visual Area MT. Vision Research, 38 (5), pp. 743-761. 1998.

[76] Golland. P., Bruckstein A. M. Motion from Color. Computer Vision and Image Understanding, 68 (3), pp. 346-362. 1997.

[77] Heitz F., Bouthemy P. Multimodal Estimation of Discontinuous Optical Flow Using Markov Random Fields. IEEE Transactions on Patterns Analysis and Machine Intelligence, 15 (12), pp. 1217-1232. 1993.

[78] Hongche Liu, Tsai-Hong Hong, Martin Herman, and Rama Chellappa. A general motion model and spatio-temporal filters for computing optical flow. International Journal of Computer Vision, 22 (2), pp. 141–172. 1997.

[79] Liu H., Hong T., Herman M. A General Motion Model and Spatio-Temporal Filters for Computing Optical Flow. Int. Journal of Computer Vision, 22 (2), pp. 141-172. 1997.

[80] Fleet D. J., Black M. J., Yacoob Y., Jepson A. D. Design and Use of Linear Models for Image Motion Analysis. Int. Journal of Computer Vision, 36 (3), pp. 171-193. 2000.

[81] Yacoob. Y. and Davis. L. S. Temporal Multi-scale Models for Flow and Acceleration. International Journal of Computer Vision, 32 (2), pp.1-17. 1999.

[82] Giaccone P. R., Jones G. A. Feed-Forward Estimation of Optical Flow. Proceedings of the IEE International Conference on Image Processing and its Applications, pp. 204-208. 1997.

[83] Giaccone P. R., Jones G. A. Segmentation of Global Motion using Temporal Probabilistic Classification. Proc. of the British Machine Vision Conference, pp. 619-628. 1998.

[84] M. Otte and H. H. Nagel. Estimation of Optical-Flow Based on Higher-Order Spatiotemporal Derivatives in Interlaced and Noninterlaced Image Sequences. Artificial Intelligence, 78 (1-2), pp. 5–43. October 1995.

[85] Nagel H. Displacement Vectors Derived from Second-Order Intensity Variations in Image Sequences. Computer Vision, Graphics and Image Processing, 21, pp. 85-117. 1983.

[86] Johnston A., McOwen P. W., Benton C. Robust Velocity Computation from a Biologically Motivated Model of Motion Perception. Proceedings of the Royal Society of London B, 266, pp. 509-518. 1999.

[87] A. Johnston, C. W Clifford. Perceived Motion of Contrast modulated Gratings: Prediction of the McGM and the role of Full-Wave rectification. Vision Research, 35, pp. 1771-1783. 1995.

[88] A. Anderson, P. W. McOwan. Humans deceived by predatory stealth strategy camouflaging motion. Proceedings of the Royal Society, B, 720, pp. 18-20. 2003.

[89] A. Johnston, C. W. Clifford. A Unified Account of Three Apparent Motion Illusions. Vision Research, 35, pp. 1109-1123. 1994.

[90] A. Johnston, P.W. McOwan, C.P. Benton. Biological computation of image motion from flows over boundaries. Journal of Physiology. Paris, 97, pp. 325-334. 2003.

[91] Campani M., Verri A. Motion Analysis from First-Order Properties of Optical Flow. CVGIP: Image Understanding, 56 (1), pp. 90-107. 1992.

[92] Bergen J. R., Burt P. J. A Three-Frame Algorithm for Estimating Two-Component Image Motion. Pattern Analysis and Machine Intelligence, IEEE T, 14 (9), pp. 886-895. 1992.

[93] Gupta N. C., Kanal L. N. 3-D Motion estimation from Motion Field. Artificial Intelligence, 78, pp. 45-86. 1995.

[94] Gupta N. C., Kanal L. N. Gradient Based Image Motion Estimation without Computing Gradients. International Journal of Computer Vision, 22 (1), pp. 81-101. 1997.

[95] Franceschini N., Pichon J. M., Blanes C. From Insect Vision to Robot Vision. Philosophical Transactions of the Royal Society of London, B, 337, pp. 283-294. 1992.

[96] Hongche Liu, Tsai-Hong Hong, Martin Herman, Ted Camus, and Rama Chellappa. Accuracy vs Efficiency Trade-offs in Optical Flow Algorithms. Computer Vision and Image Understanding, 72 (3), pp. 271–286. 1998.

[97] Hiroaki Niitsuma and Tsutomu Maruyama. Real-time detection of moving objects. Field Programmable Logic and Application, pp. 1155–1157. Springer, 2004.

[98] [online]. Xilinx. FPGA Virtex-II & Virtex-II Pro - Complete Data Sheet. http://www.xilinx.com/support/documentation/data_sheets/ds083.pdf. (Last accessed May 2014).

[99] Konstantinos Babionitakis, Gregory Doumenis, George Georgakarakos, George Lentaris, Kostantinos Nakos, Dionysios I. Reisis, Ioannis Sifnaios, and Nikolaos Vlassopoulos. A real-time motion estimation FPGA architecture. J. Real-Time Image Processing, 3 (1-2), pp. 3–20. 2008.

[100] S. Asano, Zheng Zhi Shun, and T. Maruyama. An FPGA implementation of full-search variable block size motion estimation. Field-Programmable Technology (FPT), pp. 399–402. International Conference on, December 2010.

[101] [online]. Xilinx. FPGA Virtex-5 - Data Seets Documentation. http://www.xilinx.com/support/documentation/data_sheets/ds100.pdf. (Last accessed May 2014).

[102] A. Akin, G. Sayilar, and I. Hamzaoglu. High performance hardware architectures for one bit transform based single and multiple reference frame motion estimation. Consumer Electronics, IEEE Transactions on, 56 (2), pp. 1144–1152. May 2010.

[103] B. Natarajan, V. Bhaskaran, and K. Konstantinides. Low-complexity block-based motion estimation via one-bit transforms. Circuits and Systems for Video Technology, IEEE Transactions on, 7 (4), pp. 702–706. 1997.

[104] Huong Ho, R. Klepko, Nam Ninh, and Demin Wang. A high performance hardware architecture for multi-frame hierarchical motion estimation. Consumer Electronics, IEEE Transactions on, 57 (2), pp. 794–801. 2011.

[105] Demin Wang, Liang Zhang, and André Vincent. Motion-Compensated Frame Rate Up-Conversion - Part I: Fast Multi-Frame Motion Estimation. TBC, 56 (2), pp. 133–141. 2010.

[106] José L. Núñez-Yañez, A. Nabina, E. Hung, and G. Vafiadis. Cogeneration of Fast Motion Estimation Processors and Algorithms for Advanced Video Coding. Very Large Scale Integration (VLSI) Systems, IEEE Transactions on, 20 (3), pp. 437–448. 2012.

[107] Yu-Wen Huang, Ching-Yeh Chen, Chen-Han Tsai, Chun-Fu Shen, and Liang-Gee Chen. Survey on block-matching motion estimation algorithms and architectures with new results. Journal of VLSI signal processing systems for signal, image and video technology, 42 (3), pp. 297–320. 2006.

[108] Zhaoyi Wei, Dah-Jye Lee, B. Nelson, M. Martineau, Zhaoyi Wei, Dah-Jye Lee, and M. Martineau. A Fast and Accurate Tensor-based Optical Flow Algorithm Implemented in FPGA. Applications of Computer Vision WACV '07, p. 18. IEEE Workshop on, 2007.

[109] Björn Johansson and Gunnar Farnebäck. A theoretical comparison of different orientation tensors. Proceedings SSAB02 Symposium on Image Analysis, pp. 69–73. Citeseer, 2002.

[110] G. Farnebäck. Fast and accurate motion estimation using orientation tensors and parametric motion models. Pattern Recognition Proceedings, 15th International Conference on, 1, pp. 135–139. 2000.

[111] Javier Díaz, Eduardo Ros, Rodrigo Agis, and Jose Luis Bernier. Superpipelined high-performance optical-flow computation architecture. Computer Vision and Image Understanding, 112 (3), pp. 262–273. 2008.

[112] G. Botella, A. García, M. Rodriguez, E. Ros, U. Meyer-Baese, and M.C. Molina. Robust Bioinspired Architecture for Optical-Flow Computation. VLSI Syst., IEEE Trans., 18 (4), pp. 616–629. 2010.

[113] [online]. Xilinx. FPGA Virtex-E - Field Programmable Gate Arrays. http://www.xilinx.com/support/documentation/data_sheets/ds022.pdf. (Last accessed May 2014).

[114] M.R.B. Bahar and G. Karimian. High performance implementation of the Horn and Schunck optical flow algorithm on FPGA. Electrical Engineering (ICEE), 20th Iranian Conference on, pp. 736–741. May 2012.

[115] [online]. Altera Corporation. Cyclone II FPGAs. http://www.altera.com/devices/fpga/cyclone2/cy2-index.jsp. (Last accessed May 2014).

[116] F. Barranco, M. Tomasi, J. Diaz, M. Vanegas, and E. Ros. Parallel Architecture for Hierarchical Optical Flow Estimation Based on FPGA. Very Large Scale Integration (VLSI) Systems, IEEE Transactions on, 20 (6), pp. 1058–1067. 2012.

[117] [online]. Xilinx. FPGA Virtex-4 - Family Overview. http://www.xilinx.com/support/documentation/data_sheets/ds112.pdf. (Last accessed May 2014).

[118] Tunley, H., Young, D. First order optic flow from log-polar sampled images. Computer Vision-ECCV'94, Lecture Notes in Computer Science, 800, pp. 132–137. Springer, Berlin, 1994, doi 10.1007/3-540-57956-7_14.

[119] Gokhan Koray Gultekin, Afsar Saranli, An FPGA based high performance optical flow hardware design for computer vision applications. Microprocessors and Microsystems, 37 (3), pp. 270-286. May 2013, ISSN 0141-9331, http://dx.doi.org/10.1016/j.micpro.2013.01.001.

[120] Ehsan Norouznezhad, Abbas Bigdeli, Adam Postula, and Brian C. Lovell. Object tracking on FPGA-based smart cameras using local oriented energy and phase features. Proceedings of the Fourth ACM/IEEE International Conference on Distributed Smart Cameras, ICDSC '10, pp. 33–40. ACM, New York, USA, 2010.

[121] Dennis Gabor. Theory of communication Part 1: The analysis of information. Electrical Engineers Part III: Radio and Communication Engineering, Journal of the Institution of, 93 (26), pp. 429–441.1946.

[122] M. Tomasi, M. Vanegas, F. Barranco, J. Diaz, and E. Ros. High-Performance Optical-Flow Architecture Based on a Multi-Scale, Multi-Orientation Phase-Based Model. Circuits and Systems for Video Technology, IEEE Transactions on, 20 (12), pp. 1797–1807. 2010.

[123] T. Gautama and M.M. Van Hulle. A phase-based approach to the estimation of the optical flow field using spatial filtering. Neural Networks, IEEE Transactions on, 13 (5), pp. 1127–1136. 2002.

[124] M. Tomasi, M. Vanegas, F. Barranco, J. Daz, and E. Ros. Massive Parallel-Hardware Architecture for Multiscale Stereo, Optical Flow and Image-Structure Computation. Circuits and Systems for Video Technology, IEEE Transactions on, 22 (2), pp. 282–294. 2012.

[125] Silvio P. Sabatini, Giulia Gastaldi, Fabio Solari, Karl Pauwels, Marc M. Van Hulle, Javier Diaz, Eduardo Ros, Nicolas Pugeault, and Norbert Kru¨ger. A compact harmonic code for early vision based on anisotropic frequency channels. Computer Vision and Image Understanding, 114 (6), pp. 681–699, 2010.

[126] Johny Paul, Andreas Laika, Christopher Claus, Walter Stechele, Adam El Sayed Auf, Erik Maehle. Real-time motion detection based on SW/HW-codesign for walking rescue robots. Journal of Real-Time Image Processing, 8 (4), pp. 353-368. December 2013.

[127] Komorkiewicz, Mateusz; Kryjak, Tomasz; Gorgon, Marek. Efficient Hardware Implementation of the Horn-Schunck Algorithm for High-Resolution Real-Time Dense Optical Flow Sensor. Sensors, 14 (2), pp. 2860-2891. 2014.

[128] Jean-Yves Bouguet. Pyramidal Implementation of the Affine Lucas Kanade Feature Tracker - Description of the algorithm. Intel Corporation, 2001.

[129] Simon Baker and Iain Matthews. Equivalence and efficiency of image alignment algorithms. Computer Vision and Pattern Recognition, CVPR 2001, Proceedings of the 2001, IEEE Computer Society Conference on, 1, pp. 1–1090. IEEE, 2001.

[130] G. Kiss, E. Nielsen, F. Orderud, and H.G. Torp. Performance optimization of block matching in 3D echocardiography. Ultrasonics Symposium (IUS), IEEE International, pp. 1403–1406. 2009.

[131] Jonas Crosby, Brage H. Amundsen, Torbjørn Hergum, Espen W. Remme, Stian Lange- land, and Hans Torp. 3-D Speckle Tracking for Assessment of Regional Left Ventricular Function. Ultrasound in Medicine & Biology, 35 (3), pp. 458–471. 2009.

[132] [online]. NVIDIA Corporation. Geforce GTX 285. http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-285. (Last accesse May 2014).

[133] Guangyong Zhang, Liqiang He, and Yanyan Zhang. Parallel Best Neighborhood Matching Algorithm Implementation on GPU Platform. Computer and Information Technology (CIT), IEEE 10th International Conference on, pp. 1140–1145. 2010.

[134] Zhou Wang, Yinglin Yu, and D. Zhang. Best neighborhood matching: an information loss restoration technique for block-based image coding systems. Image Processing, IEEE Transactions on, 7 (7), pp. 1056–1061. 1998.

[135] [online]. NVIDIA Corporation. Tesla Supercomputing Solutions. http://www.nvidia.com/object/tesla-supercomputing-solutions.html. (Last accessed May 2014).

[136] E. Monteiro, B. Vizzotto, C. Diniz, B. Zatt, and S. Bampi. Applying CUDA Architecture to Accelerate Full Search Block Matching Algorithm for High Performance Motion Estimation in Video Encoding. Computer Architecture and High Performance Computing (SBAC-PAD), 23rd International Symposium on, pp. 128–135. 2011.

[137] [online]. NVIDIA. Geforce GTX 480. http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-480/specifications. (Last accessed May 2014).

[138] B. Ranft, T. Schoenwald, and B. Kitt. Parallel matching-based estimation - a case study on three different hardware architectures. Intelligent Vehicles Symposium (IV), pp. 1060–1067. IEEE, 2011.

[139] J.-B. Note, M. Shand, and J.E. Vuillemin. Real-Time Video Pixel Matching. Field Programmable Logic and Applications, FPL '06, International Conference on, pp. 1–6. 2006.

[140] [online]. NVIDIA. Geforce GTX 470. http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-470. (Last accessed May 2014).

[141] Jinglin Zhang, J.-F. Nezan, and J.-G. Cousin. Implementation of Motion Estimation Based on Heterogeneous Parallel Computing System with OpenCL. High Performance Computing and Communication, IEEE 9th International Conference on Embedded Software and Systems (HPCC-ICESS), IEEE 14th International Conference on, pp. 41 –45. June 2012.

[142] [online]. AMD. Radeon HD 6870. http://www.amd.com/us/products/desktop/graphics/amd-radeon-hd-6000/hd-6870/Pages/amd-radeon-hd-6870-overview.aspx. (Last accessed May 2014).

[143] R. Rodriguez Sanchez, J.L. Martınez, G. Fernandez Escribano, J.L. Sanchez, and J.M. Claver. A Fast GPU-Based Motion Estimation Algorithm for HD 3D Video Coding Parallel and Distributed Processing with Applications (ISPA), IEEE 10th International Symposium on, pp. 166–173. 2012.

[144] E. Monteiro, M. Maule, F. Sampaio, C. Diniz, B. Zatt, and S. Bampi. Real-time block matching motion estimation onto GPGPU. Image Processing (ICIP), 19th IEEE International Conference on, pp. 1693–1696. 2012.

[145] Dung Vu, Yang Yang, and L. Bhuyan. An efficient dynamic multiple-candidate motion vector approach for GPU-based hierarchical motion estimation. Performance Computing and Communications Conference (IPCCC), IEEE 31st International, pp. 342–351. 2012.

[146] Y.L. Chan and W.C. Siu. Adaptive multiple-candidate hierarchical search for block matching algorithm. Electronics Letters, 31 (19), p. 1637. 1995.

[147] [online]. NVIDIA Corporation. GPU Tesla C250. http://www.nvidia.com/docs/IO/43395/NV_DS_Tesla_C2050_C2070_jul10_lores.pdf. (Last accessed May 2014).

[148] J. Chase, B. Nelson, J. Bodily, Zhaoyi Wei, and Dah-Jye Lee. Real-Time Optical Flow Calculations on FPGA and GPU Architectures: A Comparison Study. Field-Programmable Custom Computing Machines, FCCM '08, 16th International Symposium on, pp. 173–182. 2008.

[149] [online]. NVIDIA Corporation. GeForce 8800 gtx. http://www.nvidia.com/page/geforce_8800.html. (Last accessed 2014).

[150] Julien Marzat, Yann Dumortier, André Ducrot, et al. Real-time dense and accurate parallel optical flow using CUDA. 7th International Conference WSCG. 2009.

[151] Bernardt Duvenhage, J. P. Delport, and Jason de Villiers. Implementation of the Lucas-Kanade image registration algorithm on a GPU for 3d computational platform stabilisation. Proceedings of the 7th International Conference on Computer Graphics, Virtual Reality, Visualisation and Interaction in Africa, AFRIGRAPH '10, pp. 83–90. ACM, New York, USA, 2010.

[152] R. Phull, P. Mainali, Qiong Yang, H. Sips, and G. Lafruit. Robust Low Complexity Feature Tracking using CUDA. Signal Processing Systems (SIPS), IEEE Workshop on, pp. 362–367. 2010.

[153] P. Mainali, Qiong Yang, G. Lafruit, R. Lauwereins, and L. Van Gool. Robust low complexity feature tracking. Image Processing (ICIP), 17th IEEE International Conference on, pp. 829–832. 2010.

[154] [online]. NVIDIA Corporation. GeForce 280 gtx. http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-280. (Last accessed 2014).

[155] Rafael del Riego, Jose Otero, and Jose Ranilla. A low-cost 3D human interface device using GPU-based optical flow algorithms. Integrated Computer-Aided Engineering, 18, pp. 391–400. 2011.

[156] [online]. NVIDIA Corporation. GeForce 9500 GT. http://www.geforce.com/hardware/desktop-gpus/geforce-9500-gt. (Last accessed 2014).

[157] Robert Hegner, Ivar Austvoll, Tom Ryen, and Guido M Schuster. Efficient Implementation of Optical Flow Algorithm based on Directional Filters on a GPU Using CUDA. EUSIPCO, 2011.

[158] Robert Hegner and Guido Schuster. Efficient implementation and evaluation of methods for the estimation of motion in image sequences. PhD thesis. R. Hegner, 2010.

[159] [online]. NVIDIA. GeForce GTX 260. http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-260. (Last accessed May 2014).

[160] J. Ohmura, A. Egashira, S. Satoh, T. Miyoshi, H. Irie, and T. Yoshinaga. Multi- GPU Acceleration of Optical Flow Computation in Visual Functional Simulation. Networking and Computing (ICNC), Second International Conference on, pp. 228–234. December 2011.

[161] Manish P. Shiralkar and Robert J. Schalkoff. A self-organization based optical flow estimator with GPU implementation. Mach. Vis. Appl, 23 (6), pp. 1229–1242. 2012.

[162] T. Kohonen. The self-organizing map. Proceedings of the IEEE, 78 (9), pp. 1464–1480. 1990.

[163] F Ayuso, G Botella, C Garcia, M Prieto, F Tirado. GPU-based acceleration of bioinspired motion estimation model. Concurrency and Computation: Practice and Experience, 25 (8), pp. 1037-1056.

[164] C Garcia, G Botella, F Ayuso, M Prieto, F Tirado. Multi-GPU based on multicriteria optimization for motion estimation system. EURASIP Journal on Advances in Signal Processing, 1, pp. 1-12. 2013.

[165] K. Pauwels and M.M. Van Hulle. Realtime phase-based optical flow on the GPU. Computer Vision and Pattern Recognition Workshops, CVPRW '08, IEEE Computer Society Conference on, pp. 1–8. June 2008.

[166] Narayanan Sundaram, Thomas Brox, and Kurt Keutzer. Dense Point Trajectories by GPU-Accelerated Large Displacement Optical Flow. Kostas Daniilidis, Petros Maragos, and Nikos Paragios, editors, Computer Vision – ECCV, Lecture Notes in Computer Science, 6311, pp. 438–451. Springer, Berlin Heidelberg, 2010.

[167] T. Brox, C. Bregler, and J. Malik. Large displacement optical flow. Computer Vision and Pattern Recognition, CVPR 2009, IEEE Conference on, pp. 41–48. 2009.

[168] Pascal Gwosdek, Henning Zimmer, Sven Grewenig, Andres Bruhn, and Joachim Weickert. A Highly Efficient GPU Implementation for Variational Optic Flow Based on the Euler-Lagrange Framework. Kiriakosn. Kutulakos, editor, Trends and Topics in Computer Vision, Lecture Notes in Computer Science, 6554, pp. 372–383. Springer Berlin Heidelberg, 2012.

[169] A. Abramov, K. Pauwels, J. Papon, F. Worgotter, and B. Dellen. Real-Time Segmentation of Stereo Videos on a Portable System With a Mobile GPU. Circuits and Systems for Video Technology, IEEE Transactions on, 22 (9), pp. 1292–1305. 2012.

[170] Karl Pauwels, Norbert Krüger, Markus Lappe, Florentin Wörgötter, and Marc M Van Hulle. A cortical architecture on parallel hardware for motion processing in real time. Journal of vision, 10 (10). 2010.

[171] [online]. NVIDIA Corporation. GeForce GT 240M Specifications. http://www.geforce.com/hardware/notebook-gpus/geforce-gt-240m/specifications. (Last accessed May 2014).

[172] ISO/IEC JTC 1/SC 29/WG 11. Multiview Coding Using AVC. January 2006.

[173] Carlo Tomasi and Takeo Kanade. Detection and tracking of point features. School of Computer Science, Carnegie Mellon Univ., 1991.

[174] Francisco D Igual, Guillermo Botella, Carlos García, Manuel Prieto, Francisco Tirado. Robust motion estimation on a low-power multi-core DSP. EURASIP Journal on Advances in Signal Processing. May 2013.

[175] Hans-Hellmut Nagel and Wilfried Enkelmann. An Investigation of Smoothness Constraints for the Estimation of Displacement Vector Fields from Image Sequences. Pattern Analysis and Machine Intelligence, IEEE Transactions on, PAMI, 8 (5), pp. 565–593. 1986.

[176] Henning Zimmer, Andres Bruhn, Joachim Weickert, Levi Valgaerts, Agustın Salgado, Bodo Rosenhahn, and Hans-Peter Seidel. Complementary Optic Flow. Daniel Cremers, Yuri Boykov, Andrew Blake, and FrankR. Schmidt, editors, Energy Minimization Methods in Computer Vision and Pattern Recognition, Lecture Notes in Computer Science, 5681, pp. 207–220. Springer Berlin Heidelberg, 2009.

[177] Subramaniam B, Wu-chun Feng: The Green Index: A Metric for Evaluating System-Wide Energy Efficiency in HPC Systems. 8th IEEE Workshop on High-Performance, Power-Aware Computing (HPPAC). IEEE Computer Society, Los Alamitos, CA (USA), May 2012.

[178] Dhoot C, Mooney VJ, Chowdhury SR, Chau LP. Fault tolerant design for low power hierarchical search motion estimation algorithms. VLSI-SoC, pp. 266-271. IEEE Computer Society, Los Alamitos, CA (USA), 2011.

[179] De Vleeschouwer C, Nilsson T. Motion estimation for low power video devices. ICIP, 2, pp. 953-956. IEEE Computer Society, Los Alamitos, CA (USA), 2001.

[180] Anguita M, Díaz J, Ros E, Fernandez-Baldomero FJ. Optimization strategies for high-performance computing of optical-flow in general-purpose processors. IEEE Trans. Circuits Syst. Video Techn, 19 (10), pp. 1475–1488. 2009.

[181] Monson, J.; Wirthlin, M.; Hutchings, B.L. Implementing high-performance, low-power FPGA-based optical flow accelerators in C. Application-Specific Systems, Architectures and Processors (ASAP), IEEE 24th International Conference on , 363 (369), pp. 5-7. June 2013, doi 10.1109/ASAP.2013.6567602.

[182] Jing Hui; Wei Liming, Research on embedded vehicle image monitoring algorithms based on DSP. World Automation Congress (WAC), 1 (3), pp. 24-28. June 2012.

[183] Lalonde, J.; Laganiere, R.; Martel, L. Single-view obstacle detection for smart back-up camera systems. Computer Vision and Pattern Recognition Workshops (CVPRW), IEEE Computer Society Conference on, 1 (8), pp. 16-21. June 2012, doi 10.1109/CVPRW.2012.6238887.

[184] Ai Hong; Wang Jian; Yang Yi. Video Monitoring System of Embedded Remote Based on ARM and Fast Motion Estimation Algorithm. Instrumentation, Measurement, Computer, Communication and Control, First International Conference on, 883 (886), pp. 21-23. October 2011, doi 10.1109/IMCCC.2011.223.

[185] Guzmán, Pablo; Díaz, Javier; Agís, Rodrigo; Ros, Eduardo. Optical Flow in a Smart Sensor Based on Hybrid Analog-Digital Architecture. Sensors, 10 (4), pp. 2975-2994. 2010.

[186] Honegger, D.; Greisen, P.; Meier, L.; Tanskanen, P.; Pollefeys, M., Real-time velocity estimation based on optical flow and disparity matching. Intelligent Robots and Systems (IROS), IEEE/RSJ International Conference on, 5177 (5182), pp. 7-12. October 2012, doi 10.1109/IROS.2012.6385530.

[187] Rowekamp T, Platzner M, Peters L: Specialized architectures for optical flow computation: A performance comparison of asic, dsp, and multi-dsp. Proceedings of the 8th ICSPAT, pp. 829–833. 1997.

[188] Steimer A. Global optical flow estimation by linear interpolation algorithm on a DSP microcontroller. Master Thesis. ETH Zurich, Switzerland, October 2011.

[189] Shirai Y, Miura J, Mae Y, Shiohara M, Egawa H, Sasaki S. Moving object perception and tracking by use of dsp. Computer Architectures for Machine Perception Proceedings, pp. 251–256. IEEE Computer Society, Los Alamitos, CA (USA), http://dx.doi.org/10.1109/CAMP.1993.622479, December 1993.

[190] Horn BKP, Schunck BG. Determining optical flow. Artif. Intell, 17, pp. 185–203. http://dx.doi.org/10.1016/0004-3702%2881%2990024-2, 1981.

[191] Srinivasan MV. An image-interpolation technique for the computation of optic flow and egomotion. Biol. Cybernetics, 71, pp. 401–415. 1994.

[192] Rodríguez-Vazquez, A. The eye-RIS CMOS vision system. Analog Circuit Design: Sensors, Actuators and Power Drivers, pp. 15-32. Springer, Dordrecht, The Netherlands, 2008.

[193] Ian Kuon and Jonathan Rose. Measuring the gap between FPGAs and ASICs. Proceedings of the 2006 ACM/SIGDA, 14th international symposium on Field programmable gate arrays (FPGA '06), pp. 21-30. ACM, New York, USA, 2006, doi 10.1145/1117201.1117205. .http://doi.acm.org/10.1145/1117201.1117205.

[194] [online]. Xilinx. Improving profitability through a FPGA. http://www.xilinx.com/publications/prod_mktg/easypath-7-fpga-asic-approach.pdf. (Last accessed February 2014).

[195] Altera. Generating Functionally Equivalent FPGAs and ASICs With a Single Set of RTL and Synthesis/Timing Constraints. White paper. 2009.

[196] Leong, P.H.-W. Recent Trends in FPGA Architectures and Applications. Electronic Design, Test and Applications, 4th IEEE International Symposium on, 137 (141), pp. 23-25. DELTA, January 2008, do 10.1109/DELTA.2008.14.

[197] NEC Electronics Corporation. Gate arrays and embedded arrays. White paper, Vol. A18258EJ7V0PF. 2009.

[198] S. Warrington, S. Sudharsanan, and Wai-Yip Chan. Architecture for Multiple Reference Frame Variable Block Size Motion Estimation. Circuits and Systems, IEEE International Symposium on, pp. 2894–2897. ISCAS, May 2007.

[199] R. Verma and A. Akoglu. A coarse grained and hybrid reconfigurable architecture with flexible NoC router for variable block size motion estimation. Parallel and Distributed Processing, IEEE International Symposium on, pp. 1–8. IPDPS, April 2008.

[200] Th Zahariadis and D Kalivas. A spiral search algorithm for fast estimation of block motion vectors. Signal Processing VIII, theories and applications, Proceedings of the EUSIPCO, 96, p. 3. 1996.

[201] N. Sebastiao, T. Dias, N. Roma, P. Flores, and L. Sousa. Application Specific Programmable IP Core for Motion Estimation: Technology Comparison Targeting Efficient Embedded Co-Processing Units. Digital System Design Architectures, Methods and Tools, 11th EUROMICRO Conference on, pp. 181–188. DSD, September 2008.

[202] O. Ndili and T. Ogunfunmi. Efficient fast algorithm and FPSoC for integer and fractional motion estimation in H.264/AVC. Consumer Electronics (ICCE), IEEE International Conference on, pp. 407–408. January 2011

[203] O. Ndili and T. Ogunfunmi. Hardware-oriented Modified Diamond Search for motion estimation in H.246/AVC. Image Processing (ICIP), 17th IEEE International Conference on, pp. 749–752. 2010.

[204] S. Dhahri, A. Zitouni, and R. Tourki. A parallel processing architecture for FSS block-matching motion estimation. Systems, Signals and Devices (SSD), 8th International Multi-Conference on, pp. 1–5. March 2011.

[205] Alan A. Stocker. Analog Integrated 2-D Optical Flow Sensor. Analog Integrated Circuits and Signal Processing, 46 (2), pp. 121–138. 2006.

[206] Lajos Hanzo, Peter Cherriman, Jürgen Streit. Video Compression and Communications From Basics to H.261, H.263, H.264, MPEG4 for DVB and HSDPA-Style Adaptive Turbo-Transceivers. John Wiley & Sons, 2007.

[207] [online]. Vector video standards. http://geeks.lockergnome.com/forum/topics/is-my-monitor-full-hd. (Last accessed March 2014).

[208] [online]. MPEG4 introduction. http://mpeg4epn.wordpress.com/2010/11/25/. (Last accessed March 2014).

[209] [online]. MPEG4 motion estimation. http://www.eetimes.com/document.asp?doc_id=1275845. (Last accessed April 2014).

[210] [online]. Implementing H.264. http://www.embedded.com/design/mcus-processors-and-socs/4006615/Implementing-H-264-video-compression-algorithms-on-a-software-configurable-processor. (Last accessed March 2014).

[211] [online]. H.265 definition. http://www.streamingmedia.com/Articles/Editorial/What-Is-.../What-Is-HEVC-%28H.265%29-87765.aspx. (Last accessed February 2014).

[212] Harilaos Koumaras, Michail-Alexandros Kourtis, Drakoulis Martakos. Benchmarking the Encoding Efficiency of H.265/HEVC and H.264/AVC. Paul Cunningham and Miriam Cunningham, editors, Future Network & MobileSummit Conference Proceedings. IIMC International Information Management Corporation, 2012, ISBN 978-1-905824-30-4.

[213] Shunqing Yan, Liang Hong, Weifeng He and Qin Wang. Group-Based Fast Mode Decision Algorithm for Intra Prediction in HEVC. Eighth International Conference on Signal Image Technology and Internet Based Systems. 2012.

[214] Diego González, Guillermo Botella, Uwe Meyer-Baese, Carlos García, Concepción Sanz, Manuel Prieto-Matías, and Francisco Tirado. A Low Cost Matching Motion Estimation Sensor Based on the NIOS II Microprocessor. Sensors, 12, pp. 13126-13149. 2012, doi 10.3390/s121013126.

[215] Fermín Ayuso. Aceleración de algoritmos bioinspirados para estimación de movimiento en hardware paralelo. Ph. D. Thesis. 2013.

[216] Béatrice Pesquet-Popescu, Marco Cagnazzo, Frédéric Dufaux. Motion Estimation Techniques. TELECOM ParisTech.

[217] [online]. Block matching definition. http://es.wikipedia.org/wiki/Block_matching. (Last accessed April 2014).

[218] [online]. Search methods in motion estimation. http://blog.weisu.org/2008_12_01_archive.html. (Last accessed April 2014).

[219] Deepak Turaga, Mohamed Alkanhal. Search Algorithms for Block-Matching in Motion Estimation. Mid-Term project. 18-899. Spring, 1998.

[220] Chang-da be and Robert M. Gray. An Improvement of the Minimum Distortion Encoding Algorithm for Vector Quantization. IEEE Transactions on communications, 33 (10). October 1985.

[221] Mohammed Golam Sarwer and Q.M. Jonathan Wu. Efficient Partial Distortion Search Algorithm for block based motion estimation. Electrical and Computer Engineering, CCECE '09, Canadian Conference on, pp. 890-893. IEEE, 2009.

[222] Chok-Kwan Cheung and Lai-Man Po. Normalized Partial Distortion Search Algorithm for Block Motion Estimationy. Circuits and Systems for Video Technology, IEEE Transactions on, 10 (3), pp. 417-422. IEEE, April 2000.

[223] Yui-Lam Chan, Ko-Cheung Hui, Wan-Chi Siu Adaptive partial distortion search for block motion estimation. Multimedia and Expo, ICME '02, Proceedings of IEEE International Conference on, 1, pp. 477-480. IEEE, 2002.

[224] Chok-Kwan Cheung and Lai-Man Po. A hierarchical block matching algorithm using partial distortion measure. Circuits and Systems, ISCAS '97, Proceedings of IEEE International Symposium on, 2, pp. 1237-1240. IEEE, 1997.

[225] Chun-Ho Cheung and Lai-Man Po. A Fast Block Motion Estimation using Progressive Partial Distortion Search. Intelligent Multimedia, Video and Speech Processing, Proceedings of International Symposium on, pp. 506-509. IEEE, 2001.

[226] L.C.Manikandan, Dr. R. K. Selvakumar . A New Survey on Block Matching Algorithms in Video Coding. International Journal of Engineering Research, 3 (2), pp. 121-125. February 2014.

[227] [online]. Image sequences. http://www.cipr.rpi.edu/resource/sequences. (Last accessed May 2014).

[228] R. M. Fano, Transmission of Information. M.I.T. Press. Cambridge, 1949.

[229] J. B. Connell. A Huffman-Shannon-Fado code. Proceedings of the IEEE, 61, pp, 1046-1047. 1973.

[230] D. A. Huffman. A method for the constructions of minimum-redundancy codes. Proceedings of IRE, 20 (9), pp. 1098-1101. September 1952.

[231] D. E. Pearson. Developments in model-based video coding. Proceedings of the IEEE, 83, pp. 892-906. 5-9 December 1995.

[232] ISO/IEC JTC 1/SC 29/WG 11. Information technology – coding of moving pictures and associated audio for digital storage media at up to about 1.5 Mbits/s. Part 2: Video, Draft ISO/IEC 11172-2 (MPEG-1). ISO/IEC, Geneva, 1991.

[233] ISO/IEC JTC 1/SC 29/WG 11. ITU-T H.265, Series H: Audiovisual and multimedia systems, Infrastructure of audiovisual services – Coding of moving video. 2013. http://www.itu.int/ITU-T/recommendations/rec.aspx?rec=11885&lang=es.

[234] ISO/IEC JTC 1/SC 29/WG 11. Information technology – Generic coding of audio-visual objects. Part 2: Visual, Draft ISO/IEC 14496-2 (MPEG-4), version 1. ISO/IEC, Geneva, 1998.

[235] ISO/IEC JTC1. Coding of audio-visual objects. Part 2: Visual. April 1999.

[236] Lajos Hanzo, Peter Cherriman, Jürgen Streit. Video Compression and Communications From Basics to H.261, H.263, H.264, MPEG4 for DVB and HSDPA-Style Adaptive Turbo-Transceivers. John, Section 1.3.5, p. 12. Wiley & Sons, 2007.

[237] ISO/IEC JTC 1/SC 29/WG 11. Information technology – Generic coding of moving pictures and associated audio. Part 2: Video: Draft ISO/IEC 13818-2 (MPEG-2) and ITU-T Recommendation H.262. ISO/IEC and ITU-T, Geneva, 1994.

[238] ISO/IEC. Information technology - Coding of moving pictures and associated audio for digital storage media up to about 1.5 Mbit/s – Video. Standards Organization/International Electrotechnical (in German). International Commission, 1993.

[239] S. Emani and S. Miller. DPCM picture transmission over noisy channels with the aid of a markov model. IEEE Transactions on Image Processing, 4, pp. 1473-1481. November 1995.

[240] K. R. Rao and P. Yip. Discrete Cosine Transform – Algorithms, Advantages, Applications. Academic Press, San Diego, CA, 1990.

[241] Lajos Hanzo, Peter Cherriman, Jürgen Streit. Video Compression and Communications From Basics to H.261, H.263, H.264, MPEG4 for DVB and HSDPA-Style Adaptive Turbo-Transceivers, Section 11.2, p. 386. John Wiley & Sons, 2007.

[242] G. J. Sullivan, T. Wiegand and T. Stockhammer. Draft H.26L video coding standard for mobile applications. Proceedings of IEEE International Conference on Image Processing, 3, pp. 573-576. Thessaloniki, Greece, October 2001.

[243] CCITT H.261. Video Codec for audiovisual services at p x 64 kbit/s. 1990.

[244] ITU-T/SG15. Video coding for low bitrate communication. ITU-T Recommendation H.263, Version 1. ITU-T, Geneva, 1996.

[245] H. Graravi and A. Tabatabai. Subband coding of monochrome and color images. IEEE Transactions on Circuits and Systems, 35, pp. 207-214. February 1988.

[246] Konrad J. Estimating motion in image sequences. IEEE Signal Process Mag., 16, pp. 70-91. 1999.

[247] Sohm O.P. Fast DCT algorithm for DSP with VLIW architecture. U.S. Patent 20,070,078,921. 5 April 2007.

[248] Kappagantula S., Rao, K.-R. Motion compensated interframes image prediction. IEEE Trans. Commun., 33, pp. 1011-1015. 1985.

[249] Kuo C.-J, Yeh C.-H, Odeh S.-F. Polynomial Search Algorithms for Motion Estimation. Proceedings of the 1999 IEEE International Symposium on Circuits and Systems, pp. 813-818. Orlando, FL, USA, 11 July 2012.

[250] Zhu S., Ma K.-K. A new diamond search algorithm for fast block-matching motion estimation. IEEE Trans. Image Process., 9, pp. 287-290. 2000.

[251] Zhu S. Fast Motion Estimation Algorithms for Video Coding. M.S. thesis. Nanyang Technology University: Singapore, 1998.

[252] Bei C.-D., Gray R.-M. An improvement of the minimum distortion encoding algorithm for vector quantization. IEEE Trans. Commun., 33, pp. 1132-1133. 1985.

[253] Jain J.-R., Jain A.-K. Displacement measurement and its application in interframe image coding. IEEE Trans. Commun., 29, pp. 1799-1808. 1981.

[254] Koga T., Iinuma K., Hirano A., Iijima Y., Ishiguro T. Motion-Compensated Interframe Coding for Video Conferencing. Proceedings of the IEEE National Telecommunications Conference. New Orleans, LA, USA, 15 November 1981.

[255] Liu B., Zacarrin A. New fast algorithms for estimation of block matching vectors. IEEE Trans. Circuit. Syst. Video Technol., 3, 148-157. 1993.

[256] Li R., Zeng, B., Lious M.-L. A new three-step search algorithm for block motion estimation. IEEE Trans. Circuit. Syst. Video Technol., 3, pp. 148-157. 1993.

[257] Po L.-M., Ma W.-C. A novel four-step search algorithm for block motion estimation. IEEE Trans. Circuit. Syst. Video Technol., 6, pp. 313-317. 1996.

[258] Zhu S., Ma K.-K. A new diamond search algorithm for fast block-matching motion estimation. IEEE Trans. Image Process., 9, pp. 287-290. 2000.

[259] Béatrice Pesquet-Popescu, Marco Cagnazzo, Frédéric Dufaux. Motion Estimation Techniques, 6, pp. 33-34. TELECOM ParisTech.

[260] Th. Zahariadis, D. Kalivas. A Spiral Search Algorithm For Fast Estimation Of Block Motion Vectors. Signal Processing VIII, theories and applications, Proceedings of the EUSIPCO 96, Eighth European Signal Processing Conference, 63, p. 3. 1996.

[261] Zhu C., Lin, X., Chau, L.-P. Hexagon based search pattern for fast block motion estimation. IEEE Trans. Circuit. Syst. Video Technol., 12, pp. 349-355. 2002.

[262] Parineeth M. Reddy. Embedded systems. Resonance journal, 7 (12), pp. 20-30. 2002.

[263] Pong P. Chu. Embedded SoPC Design with Nios II Processor and VHDL Examples, Chapter 1, p. 6. Wiley, October 2011, ISBN 978-1-118-00888-1.

[264] Sukriti Jalali. Trends and Implications in Embedded Systems Development. Tata Consultancy Services Limited, 2009.

[265] [online]. Milestones in embedded systems design. http://www.embedded.com/design/prototyping-and-development/4007514/Milestones-in-embedded-systems-design. (Last accessed April 2014).

[266] [online] Emerging Trends in Embedded Systems and Applications. http://www.bvrit.ac.in/iamsocial/blog/574-emerging-trends-in-embedded-systems-and-applications. (Last accessed April 2014).

[267] [online]. Altera Corporation. Arria V SoC FPGA Hard Processor System. http://www.altera.com/devices/fpga/arria-fpgas/arria-v/hard-processor-system/arrv-soc-hps.html. (Last accessed April 2014).

[268] [online]. Xilinx. FPGA Embedded Processors. http://www.xilinx.com/products/design_resources/proc_central/resource/ETP-367paper.pdf. (Last accessed March 2014).

[269] Michael J. Flynn, Wayne Luk. Computer System Design: System-on-Chip, Chapter 3, Section 2. Wiley, October 2011, ISBN 978-0-470-64336-5.

[270] Hamblen, James O., Hall, Tyson S., Furman, Michael D. Rapid Prototyping of Digital Systems, Chapter 15. 2008.

[271] [online]. . Altera Corporation SOPC Builder User Guide. http://www.altera.com/literature/ug/ug_sopc_builder.pdf. (Last accessed April 2014).

[272] [online]. Developing and Integrating FPGA Co-processors with the TiC6X Family of DSP Processors. http://www.design-reuse.com/articles/6556/developing-and-integrating-fpga-co-processors-with-the-tic6x-family-of-dsp-processors.html. (Last accessed May 2014).

[273] [online]. Altera Corporation. Nios II Processor Reference Handbook. http://www.altera.com/literature/hb/nios2/n2cpu_nii5v1.pdf. (Last accessed May 2014).

[274] [online]. Altera Corporation. DE2 Education Board introduction. http://www.altera.com/education/univ/materials/boards/de2/unv-de2-board.html. (Last accessed May 2014).

[275] [online]. Altera Corporation. DE2 Education Board User Manual. ftp://ftp.altera.com/up/pub/Altera_Material/12.1/Boards/DE2/DE2_User_Manual.pdf. (Last accessed May 2014).

[276] [online]. Altera Corporation. DE2-115 Education Board introduction. http://www.altera.com/education/univ/materials/boards/de2-115/unv-de2-115-board.html. (Last accessed May 2014).

[277] [online]. Altera Corporation. DE2-115 Education Board User Manual. ftp://ftp.altera.com/up/pub/Altera_Material/12.1/Boards/DE2-115/DE2_115_User_Manual.pdf. (Last accessed May 2014).

[278] [online]. Altera Corporation. Embedded Peripherals IP User Guide. http://www.altera.com/literature/ug/ug_embedded_ip.pdf. (Last accessed May 2014).

[279] [online]. Nios Soft Core Embedded Processor. http://courses.cs.washington.edu/courses/cse467/08au/pdfs/lectures/13-NIOS.pdf. (Last accessed May 2014).

[280] [online]. Altera Corporation. Embedded Peripherals Handbook. http://www.ict.kth.se/courses/IL2207/0506/docs/n2cpu_nii5v3.pdf. (Last accessed May 2014).

[281] [online]. EPCS Guide. http://www.alterawiki.com/wiki/EPCS_Guide. (Last accessed May 2014).

[282] [online]. Altera Corporation. Altera Memory System Design. http://www.altera.com/literature/hb/nios2/edh_ed51008.pdf. (Last accessed May 2014).

[283] [online]. Altera Corporation. Altera Embedded News and Events. http://www.altera.com/devices/processor/news/emb-news-events.html. (Last accessed May 2014).

[284] [online]. Xilinx. MicroBlaze Soft Processor Core. http://www.xilinx.com/tools/microblaze.htm. (Last accessed May 2014).

[285] Pong P. Chu. Embedded SoPC Design with Nios II Processor and VHDL Examples, Chapter 1, pp. 1-3. Wiley, October 2011, ISBN 978-1-118-00888-1.

[286] Pong P. Chu. Embedded SoPC Design with Nios II Processor and VHDL Examples, Chapter 1, pp. 3-4. Wiley, October 2011, ISBN 978-1-118-00888-1.

[287] [online]. Xilinx. Zynq-7000 Silicon Devices. http://www.xilinx.com/products/silicon-devices/soc/zynq-7000/silicon-devices/index.htm. (Last accessed May 2014).

[288] Pong P. Chu. Embedded SoPC Design with Nios II Processor and VHDL Examples, Chapter 1, pp. 4-8. Wiley, October 2011, ISBN 978-1-118-00888-1.

[289] Hamblen, James O., Hall, Tyson S., Furman, Michael D. Rapid Prototyping of Digital Systems, Chapter 18. 2008.

[290] Jaime Roberto Ticay Rivas. FPGA embedded Soft and Hard IP cores. December 2013.

[291] [online]. Altera Corporation. Dual-Core ARM Cortex-A9 MPCore Processor. http://www.altera.com/devices/processor/arm/cortex-a9/m-arm-cortex-a9.html. (Last accessed May 2014).

[292] [online]. Altera Corporation. Cyclone V SoC Hard Processor System. http://www.altera.com/devices/fpga/cyclone-v-fpgas/hard-processor-system/cyv-soc-hps.html. (Last accessed May 2014).

[293] [online]. LEON4 Processor. http://www.gaisler.com/index.php/products/processors/leon4. (Last accessed May 2014).

[294] [online]. OpenRISC project. http://opencores.org/or1k/Main_Page. (Last accessed May 2014).

[295] [online]. Altera Corporation. Nios II C2H Compiler User Guide. http://www.altera.com/literature/ug/ug_nios2_c2h_compiler.pdf. (Last accessed May 2014).

[296] [online]. Altera Corporation. Optimizing Nios II C2H Compiler Results. http://www.altera.com/literature/hb/nios2/edh_ed51005.pdf. (Last accessed May 2014).

[297] Diego González, Guillermo Botella, Uwe Meyer-Baese, Carlos García, Concepción Sanz, Manuel Prieto-Matías, and Francisco Tirado. A Low Cost Matching Motion Estimation Sensor Based on the NIOS II Microprocessor. Sensors, 12 (10). 2012.

[298] [online]. Altera Corporation. Avalon Interface Specifications. http://www.altera.com/literature/manual/mnl_avalon_spec.pdf. (Last accessed May 2014).

[299] [online]. Stephen A. Edwards. Altera's Avalon Communication Fabric. http://www.cs.columbia.edu/~sedwards/classes/2012/4840/avalon.pdf. (Last accessed May 2014).

[300] [online]. Automated Generation of Hardware Accelerators With Direct Memory Access From ANSI/ISO Standard C Functions. http://www.altera.com/literature/wp/wp-aghrdwr.pdf. (Last accessed May 2014).

[301] [online]. Preprocessor directives. http://www.cplusplus.com/doc/tutorial/preprocessor/. (Last accessed May 2014).

[302] Brian W. Kernighan, Dennis M. Ritchie. The C programming language. Prentice-Hall, 2012, ISBN 0131103628.

[303] [online]. Altera Corporation. Using the Nios II Integrated Development Environment. http://www.altera.com/literature/hb/nios2/n2sw_nii52002.pdf. (Last accessed May 2014).

[304] [online]. Altera Corporation. Altera Software Installation and Licensing. http://www.altera.com/literature/manual/quartus_install.pdf. (Last accessed May 2014).

[305] [online]. The C Preprocessor. http://gcc.gnu.org/onlinedocs/cpp/. (Last accessed May 2014).

[306] [online]. GCC compiler options. http://gcc.gnu.org/onlinedocs/gcc/Overall-Options.html. (Last accessed May 2014).

[307] [online]. Xilinx. FPGA (Field Programable Gate Array) defintion. http://www.xilinx.com/training/fpga/fpga-field-programmable-gate-array.htm. (Last accessed May 2014).

[308] Ashenden, P.J. VHDL standards. IEEE Des. Test Comput, 18, pp. 122–123. 2001.

[309] Deutschmann, R.; Koch, C. An Analog VLSI Velocity Sensor Using the Gradient Method. Proceedings of the IEEE International Symposium on Circuits and Systems. Monterey, CA, USA, 31 May 1998.

[310] Stocker, A.-A; Douglas, R.-J. Analog Integrated 2D Optical Flow Sensor with Programmable Pixels. Proceedings of the IEEE International Symposium on Circuits and Systems. Vancouver, BC, USA, 23 May 2004.

[311] Niitsuma, H.; Maruyama, T. Real-Time Detection of Moving Objects. Proceedings of the IEEE International Conference on Field Programmable Logic and Applications, pp. 1153-1157. Leuven, Belgium, 30 August 2004.

[312] Im, Y.L.; Il-Hyun, P.; Dong-Wook, L.; Ki-Young, C. Implementation of the H.264/AVC Decoder Using the Nios II Processor. http://www.altera.com/literature/dc/1.5-2005_Korea_2nd_SeoulNational-web.

[313] [online]. Anafocus, Leading on-chip vision solutions. http://www.anafocus.com. (Last accessed May 2014).

[314] Botella, G.; Meyer-Baese, U.; Garcia, A. Bio-inspired robust optical flow processor system for VLSI implementation. Electron. Lett., 45, pp. 1304–1305. 2009.

[315] Po, L.-M.; Ma, W.-C. A novel four-step search algorithm for fast block motion estimation. IEEE Trans. Circuit. Syst. Video Technol., 6, pp. 313–317. 2009.

[316] Liu, L.-K.; Feig, E. A block-based gradient descent algorithm for fast block motion estimation in video coding. IEEE Trans. Circuit. Syst. Video Technol., 6, pp. 419–422. 1996.

[317] [online]. Xilinx. Datasheet XC2V6000-5FF1152I-Virtex-II 1.5V Field-Programmable Gate Arrays. http://www.alldatasheet.com/datasheet-pdf/pdf/97992/XILINX/ XC2V6000-5FF1152I.html. (Last accessed May 2014).

[318] [online]. Altera Corporation. Nios II C-to-Hardware Acceleration Compiler. http://www.altera.com/devices/processor/nios2/tools/c2h/ni2-c2h.html. (Last accessed May 2014).

[319] [online]. Altera Corporation. Nios II Custom Instruction User Guide. http://www.altera.com/literature/ug/ug_nios2_custom_instruction.pdf. (Last accessed May 2014).

[320] Diego González, Guillermo Botella, Carlos García, Manuel Prieto and Francisco Tirado. Acceleration of block-matching algorithms using a custom instruction-based paradigm on a Nios II microprocessor. EURASIP Journal on Advances in Signal Processing,118. 2013, doi 10.1186/1687-6180-2013-118.

[321] [online]. Altera Corporation. Creating Multiprocessor Nios II Systems Tutorial. http://www.altera.com/literature/tt/tt_nios2_multiprocessor_tutorial.pdf. (Last accessed May 2014).

[322] [online]. Altera Corporation. Instantiating the Nios II Processor. http://www.altera.com/literature/hb/nios2/n2cpu_nii51004.pdf. (Last accessed May 2014).

[323] [online]. C Startup. http://www.bravegnu.org/gnu-eprog/c-startup.html. (Last accessed May 2014).

[324] H. Haussecker and P. Geissler. Handbook of computer vision and applications, 3. Academic Press, 1999.

[325] Kuglin, C., Hines, D. The phase correlation image alignment method. Proceedings of the IEEE 1975 International Conference on Systems, Man and Cybernetics, pp. 163–169. 1975.

[326] Thomas, G. Television motion measurement for DATV and other applications. British Broadcasting Corporation (BBC) Department of Research, 1987.

[327] Vlachos, T., Thomas, G. Motion estimation for the correction of twin-lens telecine flicker. Proceedings of the International Conference on Image Processing, 1996, 1, pp. 109–112. 1996, doi:10.1109/ICIP.1996.559444.

[328] Liang, Y. Phase-correlation motion estimation. EE392J Project Report. 2000.

[329] Fitch, A., Kadyrov, A., Christmas, W., Kittler, J. Fast robust correlation. IEEE Trans. Image Process, 14 (8), pp. 1063–1073. 2005.

[330] Huber, P. Robust Statistics. Wiley, New York, 1981.

[331] Chou, Y.M., Hang, H.M. A new motion estimation method using frequency components. J. Vis. Commun. Image Represent. 8 (1), pp. 83–96. 1997, ISSN 1047-3203, doi:10.1006/jvci.1997.0343.

[332] Kilthau, S.L., Drew, M.S., Mo¨ller, T. Full search content independent block matching based on the fast fourier transform. ICIP, 1, pp. 669–672. 2002.

[333] Essannouni, F., Thami, R.O.H., Aboutajdine, D., Salam, A. Fast exhaustive block-based motion vector estimation algorithm using FFT. Arab. J. Sci. Eng., 32 (1), pp. 61–74. 2007.

[334] Meyer-Baese, U., Vera, A., Meyer-Baese, A., Pattichis, M., Perry, R. Discrete wavelet transform FPGA design using Mat-Lab/Simulink. Proc. SPIE 6247, Independent Component Analyses, Wavelets, Unsupervised Smart Sensors, and Neural Networks IV, 624703. 2006.

[335] VSI Alliance. Intellectual property protection: schemes, alternatives and discussion. Version 1.1. (IPPWP1 1.1).

[336] Lie, D., Mitchell, M., Thekkath, C., Horowitz, M. Specifying and verifying hardware for tamper-resistant software. Proceedings of the 2003 Symposium on Security and Privacy, pp. 166, 177. 2003, doi:10.1109/SECPRI.2003.119933.

[337] Harper, S., Athanas, P. A security policy based upon hardware encryption. Proc. Int. Conf. Syst. Sci., 7, 1–8. 2004.

[338] Goldstein, H. The secret art of chip graffiti. IEEE Spectr. 39 (3), pp. 50–55. 2002.

[339] Castillo, E., Parrilla, L., García, A., Lloris, A., Meyer-Baese, U. IPP watermarking technique for IP core protection on FPL devices. Proceedings of 16th International Conference on Field Programmable Logic and Applications FPL'2006, pp. 487–492. 2006.

[340] Castillo, E., Meyer-Baese, U., Garcia, A., Parrilla, L., Lloris, A. IPP@HDL: efficient intellectual property protection scheme for IP cores. IEEE Trans. Very Large Scale Integr. Syst., 15 (5), pp. 578–591. 2007.

[341] Castillo, E., Parrilla, L., Garcia, A., Meyer-Baese, U., Botella, G., Lloris, A. Automated signature insertion in combinational logic patterns for HDL IP core protection. In: 2008 4th Southern Conference on Programmable Logic, pp. 183, 186. 2008, doi:10.1109/SPL.2008.4547753.

[342] Barak, B., Goldreich, O., Impagliazzo, R., Rudich, S., Sahai, A., Vadhan, S., Yang, K. On the (im)possibility of obfuscating programs. J. ACM., 59 (2), p. 48. Art. No. 6. 2012, doi:10.1145/2160158.2160159.

[343] Vera, A., Meyer-Baese, U., Pattichis, M. An FPGA-based rapid prototyping platform for wavelet coprocessors. Proc. SPIE 6576, Independent Component Analyses, Wavelets, Unsupervised Nano-Biomimetic Sensors, and Neural Networks V, 657615. 2007, doi:10.1117/12.720134.

[344] Meyer-Baese, U., Botella, G., Mookherjee, S., Castillo, E., García, A. Energy optimization of application-specific instruction-set processors by using hardware accelerators in semicustom ICs technology. Microprocess. Microsyst, 36 (2), pp. 127–137. 2012, doi:10.1016/j.micpro.2011.06.003.

[345] Meyer-Baese, U., Botella, G., Castillo, E., Garcıía, A. A balanced HW/SW teaching approach for embedded microprocessors. Int. J. Eng. Educ., 26 (3), pp. 584–592. 2010.

[346] Meyer-Baese, U., Wolf, S., Taylor, F. Accumulator-synthesizer with error-compensation. IEEE Trans. Circuits Syst. II, 45 (7), pp. 885–890. 1998.

[347] Meyer-Baese, U., Rao, S., Ramírez, J., García, A. Cost-effective Hogenauer cascaded integrator comb decimator filter design for custom ICs. IEE Electron. Lett. 41 (3), pp. 66–67. 2005.

[348] Gueguen, J., Bricaud, P. Applying the OpenMORE assessment program for IP cores. Proceedings of the IEEE 2000 First International Symposium on Quality Electronic Design, 2000. ISQED 2000, pp. 379–381. 2000. doi:10.1109/ISQED.2000.838900.

[349] Collberg, C., Thomborson, C., Low, D. A taxonomy of obfuscating transformations. Technical Report #148. Dept. CS, Uni. Of Auckland, New Zealand. 1997.

[350] [online]. ANSI. The programming language C, sec. 6.4.2 identifiers. http://www.open-std.org/jtc1/sc22/wg14/. (Last accessed May 2014).

[351] VHDL language reference manual. IEEE standard 1076 (2008), IEEE Explore, Section 15: lexical elements. 2008.

[352] Verilog. IEEE standard 1364, Chapter 3: lexical conventions. 2005.

[353] Meyer-Baese, U. Digital signal processing with field programmable gate arrays, 3rd edn. Springer, Berlin, 2007.

[354] Meyer-Baese, U. Fast Digital Signal Processing (Schnelle digitale Signalverarbeitung), 364 pages (in German). ISBN: 3-540-67662-7. Springer, Berlin, 2000.

[355] [online]. ODROID-XU+E platform. http://hardkernel.com/main/products/prdt_info.php?g_code=G137463363079. (Last accessed June 2014).

[356] [online]. Meyer-Baese, U. 128-Length obfuscator tool set. http://www.eng.fsu.edu/~umb/04.htm. (Last accessed March 2014).

.